TABLE INPUT METHOD

TABLE INPUT METHOD

A TOOL FOR THE CONSTRUCTION OF TABULAR EXPRESSIONS IN THE TABLE TOOL SYSTEM

by

JAROSLAW G. KOWALIK, B.MATH

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree

Master of Science

McMaster University

Hamilton, Ontario

© Copyright by Jaroslaw G. Kowalik, April 2002

MASTER OF SCIENCE (2002) McMASTER UNIVERSITY
(Computer Science) Hamilton, Ontario, Canada

TITLE: Table Input Method - A Tool for the Construction of Tabular Expressions in the Table Tool System

AUTHOR: Jaroslaw[1] Grzegorz Kowalik, B.Math (University of Waterloo)

SUPERVISOR: Dr. David Lorge Parnas

NUMBER OF PAGES: xii, 144

[1] Jaroslaw is a Polish name of which the short form is Jarek. Most people know me as Yarek, which is the English phonetic equivalent of Jarek.

Abstract

Compared to other means, mathematical expressions most precisely represent the complex relations that are used to specify and document software systems, however, they can be complex. In order to simply the presentation of mathematical expressions, Dr. Parnas et al. proposed the use of *tabular expressions* [8][10][11].

To create correct tabular expressions quickly, an effective editor is required. This paper proposes thirteen criteria to evaluate the effectiveness of tabular expression editors. Six common editors are evaluated benchmarked against these thirteen criteria. None of the existing surveyed editors however meet all the criteria. To prove these thirteen criteria can be fulfilled a new editor, Table Input Method (TIM), was created by the author. This paper illustrates how TIM successfully meets all the established criteria.

Acknowledgements

This thesis was written with advice and support from my professors and friends. It is impossible to list here all those who helped sustain me during the coding, writing and revising, and I apologize in advance for any omissions.

Dr. David L. Parnas was instrumental in suggesting ways to approach the research. He also tirelessly provided me with valuable feedback encouraging me to refine my work.

Other helpful and thoughtful advisors were Dr. Bob Barber, Dr. Bill Farmer and Dr. Martin von Mohrenschildt. Their valuable suggestions are greatly appreciated. Dr. von Mohrenschildt's configurable Grand Table Interface formed a basis for Table Input Method user interface design. Sairah Khoker, senior student in Software Engineering programme, did a great job in testing the software.

Special thanks should also go to Sitraka Inc., my long time employer, for the support and encouragement, and for providing quality software components that made the implementation of TIM that much easier. I would like to thank my colleagues who have so enthusiastically supported my work.

I am deeply grateful to my family for believing in me. I am indebted to my uncle Anatol Kark for telling me that I did not know much about software engineering several years ago, goading me to learn more.

And finally, thanks to my wife Ming for her love, encouragement and support that has made it all meaningful and worthwhile, and to Lulu for her unfailing companionship and curiosity that made the long nights bearable.

TABLE OF CONTENTS

Abstract ..... **iii**

Acknowledgements ..... **iv**

1 Introduction **1**

1.1 The Importance of Documentation in the Software Design Process 1

1.2 Use of Mathematics for Software Documentation 1

1.2.1 Tabular Expression Example 3

1.2.2 Use of Tabular Expressions in Software Documentation 4

1.3 TTS – The Software Design Documentation Production Tool 5

1.3.1 Tabular Expressions and Table Holder 6

1.3.2 Symbols 7

1.4 Need for a Tabular Expression Editor 7

1.5 Criteria for an Effective Tabular Expression Editor 7

1.6 An Effective Tabular Expression Editor 9

1.7 The Importance of an Effective Table Editor 12

2 Survey of Expression and Table-content Editing Applications **13**

2.1 Introduction 13

2.2 Microsoft Excel 13

2.3 Microsoft Word 15

2.3.1 Microsoft Word Equation Editor 15

2.3.2 Microsoft Word Tables 17

2.4 Adobe FrameMaker 18

2.4.1 FrameMaker Equation Editor 18

2.4.2 FrameMaker Tables 20

2.5 Interleaf 21

2.5.1 Interleaf Equation Editor 21

2.5.2 Interleaf Table Editor 22

2.6 Grand Table Interface 23

2.7 Table Construction Tool 25

3 Criteria for an Effective Tabular Expression Editor **27**

4 Characteristics of the Surveyed Tools 33
4.1 Introduction 33
4.2 Microsoft Excel 33
4.3 Microsoft Word 34
4.4 FrameMaker 35
4.5 Interleaf 35
4.6 Grand Table Interface 36
4.7 Table Construction Tool 36
5 TCT – Existing Tabular Expression Editor 37
5.1 Notation 37
5.2 Understanding Expression Evaluation 39
5.3 Expression Construction 40
5.3.1 Tables 42
5.4 Correct Expressions, Verification and Error Analysis 43
5.5 Lack of Extensibility 43
6 TIM – an Effective Tabular Expression Editor 45
6.1 Correct Expressions in TIM 46
6.2 Verification and Error Analysis in TIM 47
6.3 Free Input with Flexible Validation and Automatic Corrections in TIM 47
6.4 Choice of Preferred Notation in TIM 47
6.5 Expression State Indicators in TIM 49
6.6 Configurable Table Types in TIM 49
6.7 Insertion and Deletion of Cells in TIM 49
6.8 Editing Operations in TIM 50
6.9 Allow Using and Defining New Symbols in TIM 51
6.10 Tables in Tables in TIM 51
6.11 Error Correction in TIM 53
6.12 Extensibility and the Ability to Configure in TIM 53
6.13 Expression Helpers in TIM 55
7 TTS Environment, Challenges and Solutions 57
7.1 TTS Environment – An X/Motif based framework 57
7.2 Challenges 58
7.2.1 Expanding GTI is not feasible 58

7.2.2 Input String Requirements ... 58
7.2.3 Limitations of the X/Motif Text Widget ... 60
7.2.4 Input Parsing ... 61
7.2.5 Ability to Display Tables Efficiently ... 61
7.3 Solutions to the Challenges ... 61
7.3.1 Yudit – a Portable Locale-free Unicode Editor ... 62
7.3.2 Translating Unicode to Motif Strings ... 62
7.3.3 XRT/table – a Widget for Displaying Tables in X/Motif ... 62
7.3.4 Java TTS Expression Parsers ... 63
8 Future work ... 65
8.1 Evaluation by Users ... 65
8.2 Finer Control Over Error Correction ... 65
8.3 Keyboard Shortcuts to Activate Actions ... 65
8.4 Improve Expression State Indicators ... 66
8.5 Migrate Utilities into the TTS Framework ... 66
8.6 Modularization and Code Cleanup ... 66
9 Conclusions ... 67
9.1 Effective Tabular-Expression Editors – Design Principle Tradeoffs ... 67
9.2 Fulfilling All Criteria is Feasible, Balancing Is Needed. ... 69
9.3 Limitations inherent to TTS ... 70
9.4 TTS is not an implementation medium ... 71
9.5 Tabular Notation – It Would Have Been Beneficial ... 71
References ... 73
Appendix A: Table Input Method User's Manual ... 75
A.1 Typographical Conventions Used in this Manual ... 75
A.2 Unicode Fonts Are Required ... 75
A.3 Java 2 is required ... 76
A.4 Getting Started ... 76
A.4.1 Editing a New Expression ... 77
A.4.2 Expression Symbols ... 79
A.4.3 Rendering expressions ... 80
A.5 TIM Anatomy ... 81
A.5.1 Menu ... 82
A.5.2 Toolbar ... 84
A.5.3 Table Area ... 84

A.5.4 Input Area 87
A.5.5 Symbol Information Area 88
A.5.6 Status Bar 91
A.5.7 Syntax Tree display tool 92
A.6 Typing expression text 93
A.6.1 Typing a key sequence 93
A.7 Configuration and Advanced features 94
A.7.1 Defining a Custom Keymap 95
A.7.2 Changing TTS fonts 98
A.7.3 Specifying New Table Templates 99
A.7.4 Unicode Character Mapping 103
A.7.5 Symbol Property Type (Info Class) Display Formats 104
A.7.6 Selecting Symbol Info Classes to Display 106
A.7.7 Auto-Parse Configuration 107
Appendix B: Writing a Custom TTS Parser 109
B.1 Parser Specification File Format 110
B.1.1 Scanner Declarations 110
B.1.2 Parser Productions 111
B.2 Custom Error Correction 114
B.3 Specifying Location of Parsers 114
Appendix C: Translating Stored Expressions to Strings 115
Appendix D: TTS modifications to support TIM 117
D.1 Unicode – the new symbol class 117
D.2 Introduction of Placeholder symbol Tag 117
D.3 C++ support – modifications to TTS tool public header files 117
Appendix E: TTS-input Keymap 119
Appendix F: TTS-Unicode Keymap 123
Appendix G: Useful Unicode Symbols 125
Appendix H: Standard TIM Configuration File 131

LIST OF FIGURES

Figure 1: Yarek's Wonderful PC Hog installation dialog 2
Figure 2: Description of install wizard execution using tabular expression notation 3
Figure 3: Description of install wizard execution using traditional notation 4
Figure 4: TTS Design Schematic 6
Figure 5: A syntax tree for expression a+b=c 6
Figure 6: Constructing tabular expressions in TIM 11
Figure 7: Word Equation Editor 16
Figure 8: Word Equation Editor - completed equation 16
Figure 9: Word – a table in a table 17
Figure 10: Equation palette in FrameMaker 19
Figure 11: Editing equations in FrameMaker 20
Figure 12: Creating tables in FrameMaker 20
Figure 13: Equation Editor in Interleaf 21
Figure 14: Editing a table in Interleaf 22
Figure 15: GTI – general tabular form 23
Figure 16: GTI Table Editor 24
Figure 17: TCT - editing cell (1,1) in grid G (i.e. 3<1,1>) 25
Figure 18: Copying and pasting range of cells from Table A to Table B 30
Figure 19: TCT notational aid – underscored sub-expression 39
Figure 20: Entering sub-expressions in TCT 42
Figure 21: Table Input Method - editing tables 45
Figure 22: TIM - Insert row before menu item. 50
Figure 23: Editing table's inner <2D Normal F> table 52
Figure 24: Typing in the ∀ symbol using a keystroke sequence `uni<esc>` 54

Figure 25: Editing a symbol 55
Figure 26: Syntax Tree Dialog 56
Figure 27: Table Legend Dialog 56
Figure 28: Selecting a tool in TTS 77
Figure 29: New expression type selection dialog 78
Figure 30: Components of TIM 81
Figure 31: Table Legend dialog for a Normal 2D table 86
Figure 32: Parsing error message dialog 88
Figure 33: Editing new symbol properties 89
Figure 34: Status bar 91
Figure 35: Status bar fields 91
Figure 36: Syntax tree for expression 1+count(x, A, lower+1, upper) 92
Figure 37: Typing a sequence for universal symbol and the resulting ∀ symbol 93
Figure 38: Keymap file example 97
Figure 39: Sections and variables needed for configuring custom keymap 98
Figure 40: Adding the Nice-input keymap to TIM configuration file 98
Figure 41: Matching row and column count for neighbouring grids 101
Figure 42: Template configuration example 102
Figure 43: Sample TIM Unicode Map configuration file section 104
Figure 44: Editor for an enum type info class (Tag) 105
Figure 45: Sample *TIM Info Display Format* configuration file section 105
Figure 46: Sample configuration for which info classes to display 106
Figure 47: Auto-parse configuration variables 107
Figure 48: Sample auto-parse configuration 107
Figure 49: Sample of TIM parser rules 113
Figure 50: Sample input parser configuration 114
Figure 51: Parser-driven algorithm for expression input string reconstruction 116

LIST OF TABLES

Table 1: Characteristics of Surveyed Applications 9
Table 2: Characteristics of Microsoft Excel 34
Table 3: Characteristics of Microsoft Word 34
Table 4: Characteristics of FrameMaker 35
Table 5: Characteristics of Interleaf 35
Table 6: Characteristics of GTI 36
Table 7: Characteristics of TCT 37
Table 8: Examples of expression notation 38
Table 9: Changing expression a*b into a*b+c*d in TCT 41
Table 10: Characteristics of TIM 46
Table 11: Basic symbol property classes 79
Table 12: New Symbol Properties 90
Table 13: Description of status bar field contents 91
Table 14: TIM configuration file sections 95
Table 15: Keymap file syntax 96
Table 16: Keymap file - special key hex codes 96
Table 17: Keymap file - special key sequences for function keys 97
Table 18: Valid Font-Face values 99
Table 19: Sample *TIM TTS Fonts* configuration file section 99
Table 20: table template properties 100
Table 21: Valid CCG values. 100
Table 22: Info class format attributes 104
Table 23: Info class editor by type. 105
Table 24: Variables used for component info class symbol display configuration 106
Table 25: Parser file sections 110
Table 26: Meaning of parser rule constructs 112

1 Introduction

1.1 The Importance of Documentation in the Software Design Process

Software plays an important role in our everyday lives. It has become an indispensable part of every vital and critical system in our society. Software is used in cars, hospitals, courts, and nuclear power-generating stations [1]. We depend on software's proper functioning for our safety, health, and well-being [2].

While an increasing level of attention is being focused on the hazards of faulty software [4] [5], those who design and write software must take precautions to ensure that their software is trustworthy. Consequently, creating reliable and correct software has become a critical issue and the need for the precise documentation of software is growing.

Creating software is not currently an engineering discipline but it should be. Engineers from classical disciplines such as mechanical and civil engineering design their products by documenting them and analyzing their design before building them. Unfortunately, in practice, some software developers frequently document their product *after* creation, if they do so at all. This leaves no room for verification or second opinions about the particular fitness of the product [6].

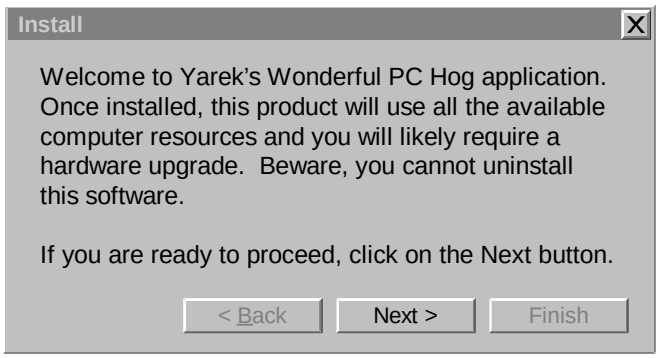
1.2 Use of Mathematics for Software Documentation

To ensure the quality of software, developers must be able to describe the behaviour of software precisely. Software documentation should be easily understood by the designers, programmers and the domain experts who are familiar with the application area. It should stand up to scrutiny by the most thorough analysis and form a basis for

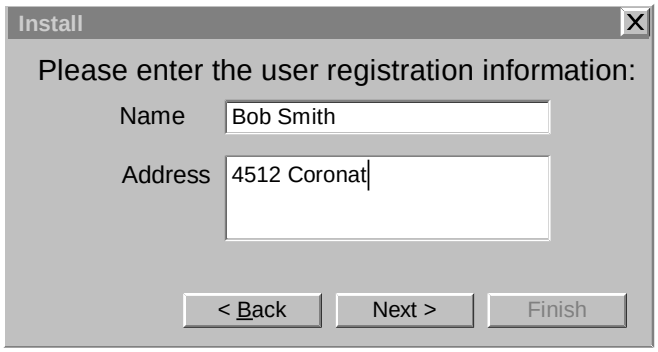
carefully planned testing of the product. As such, documentation can serve as the software design medium [7].

Mathematics is the most precise form of documentation, and as such it can be used for documenting computer systems [8] [9]. Frequently complex expressions can be represented in a more readable way using tabular expressions whose components are logical expressions and terms [10]. The meaning of the tabular expressions can be defined by rules for translating these tables into more conventional expressions. The formal semantics of the tabular expressions are described in [11].

Page 1:


Install
Welcome to Yarek's Wonderful PC Hog application. Once installed, this product will use all the available computer resources and you will likely require a hardware upgrade. Beware, you cannot uninstall this software.
If you are ready to proceed, click on the Next button.
< Back
Next >
Finish

Page 2:



Page 3:

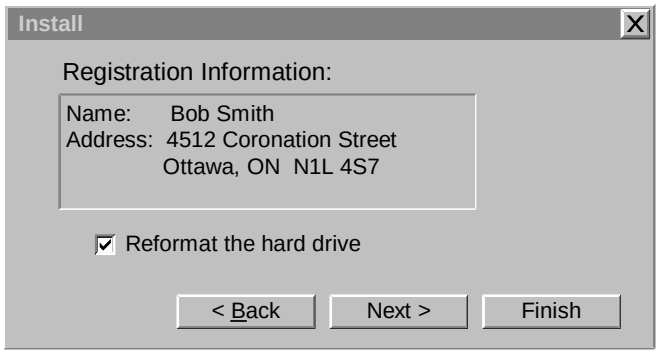


Figure 1: Yarek's Wonderful PC Hog installation dialog

1.2.1 Tabular Expression Example

To illustrate the use of tabular expressions consider the install wizard dialog for Yarek's Wonderful PC Hog application (Figure 1). The wizard is composed of three pages. The first page contains the introduction. On the second page the user is required to enter a name and address, without which progress to the next wizard page is not allowed. The final page summarises the registration information and presents the user with an option to perform a special task.

On each page the user can only take valid actions initiated by pressing the appropriate button. When the user presses the **Next >** button, the wizard proceeds to the next page. Pressing the **< Back** button will move the wizard to back the previous page, and the **Finish** button completes the registration process, dismisses the wizard dialog, optionally reformats the hard drive, and installs the program.

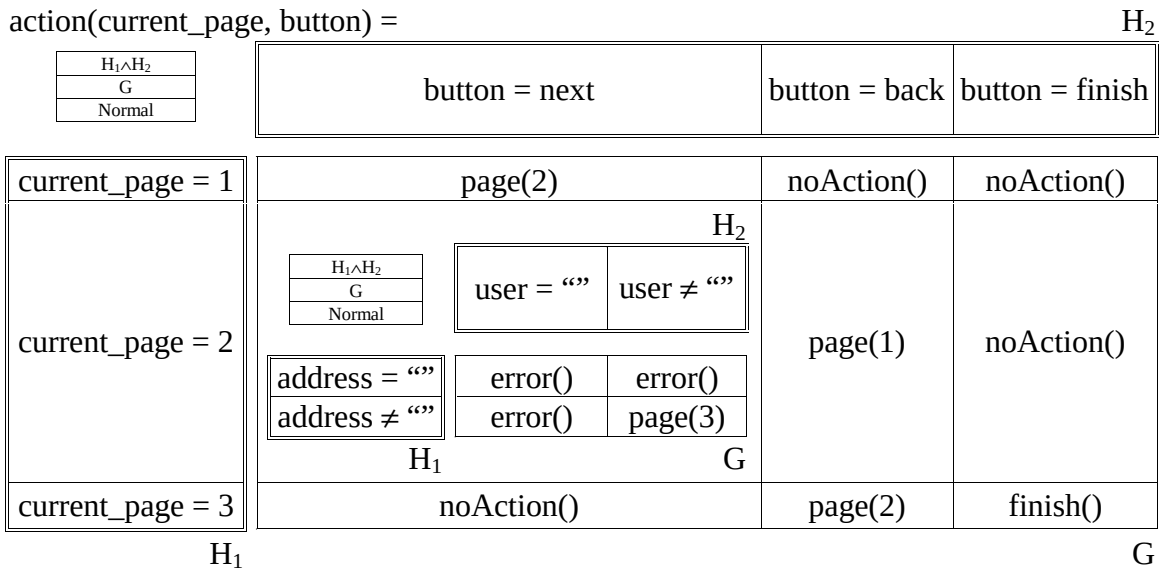


Figure 2: Description of install wizard execution using tabular expression notation

The installation process can be described more succinctly in tabular form, as shown in Figure 2. The row labels of the table show the value of the *current_page* variable, which corresponds to the current wizard page shown. The column labels show the value of the *button* variable, which corresponds to the button pressed by the user. The

cells of the table indicate what actions are to be taken given the page shown and button pressed.

The function call *page(n)* means move to page *n*, *finish()* means finish the install, *error()* means flag an error to the user, and *noAction()* means no action is to be taken (in such situation the corresponding button can be disabled).

In this example the cell in the second row of the first column contains another table. This inner table shows that moving from page 2 to page 3 is permitted only when both the name and the address fields are not empty. Otherwise, an error is flagged.

A definition of the wizard dialog action function using a more traditional notation is shown in Figure 3. In comparison, the tabular representation of the action function is easier to read and inspect than the more traditional description.

action(current_page, button) =
⎧ page(1) if (current_page = 2) ∧ (button = back)
⎪ page(2) if ((current_page = 1) ∧ (button = next)) ∨ ((current_page = 3) ∧ (button = back)
⎪ page(3) if (current_page = 2) ∧ (button = next) ∧ ((address ≠ “”) ∧ (user ≠ “”))
⎪ finish() if (current_page = 3) ∧ (button = finish)
⎪ error() if (current_page = 2) ∧ (button = next) ∧ (((address = “”) ∧ (user = “”)) ∨
⎨ ((address = “”) ∧ (user ≠ “”)) ∨
⎪ ((address ≠ “”) ∧ (user = “”)))
⎪ noAction() if ((current_page = 1) ∧ (button = back) ∨
⎪ ((current_page = 1) ∧ (button = finish) ∨
⎪ ((current_page = 2) ∧ (button = finish) ∨
⎩ ((current_page = 3) ∧ (button = next)

Figure 3: Description of install wizard execution using traditional notation

1.2.2 Use of Tabular Expressions in Software Documentation

The above example demonstrates that tabular expressions can be used for documenting software. In particular, tabular expressions can be used at each of the following steps of software development:

- Specification of product requirements
- Documentation of designs

- Product analysis
- Product simulation
- Computation of product properties
- Implementation
- Product testing
- Product maintenance and revision

Precise mathematical software documentation can be used to analyse the software product more thoroughly and effectively. It can also be used to form a basis for effective design reviews, to verify that the design fulfills the requirements, to guide developers during implementation, to check that implementation of the software matches the design, to validate testing by automatically generating test oracles [12], and more.

1.3 TTS – The Software Design Documentation Production Tool

The syntax of tabular expressions is not compatible with existing document production tools such as word processors, mathematical expression editors and spreadsheets. The Software Engineering Research Group (SERG) at McMaster University has created the Table Tool System (TTS) as a vehicle for developing techniques and tools to facilitate the production of software design documentation.

The goal of the TTS project is to develop an integrated, extensible system of tools that work together to facilitate the use of tabular expressions in computer system documentation. TTS is prepared for future growth by using an application framework (Figure 4). Design decisions and elements common to all TTS components are part of this framework, facilitating the addition or deletion of tools and/or development of new applications.

Each tool in TTS fulfills a different function, and each function works together to manipulate, present and verify the final document. Some of the existing tools are: the Table Printing Tool [14], the Table Construction Tool [15], the Inversion/Normalization Tool [16], the Code Generator [12], the Symbol Editor [13], and others.

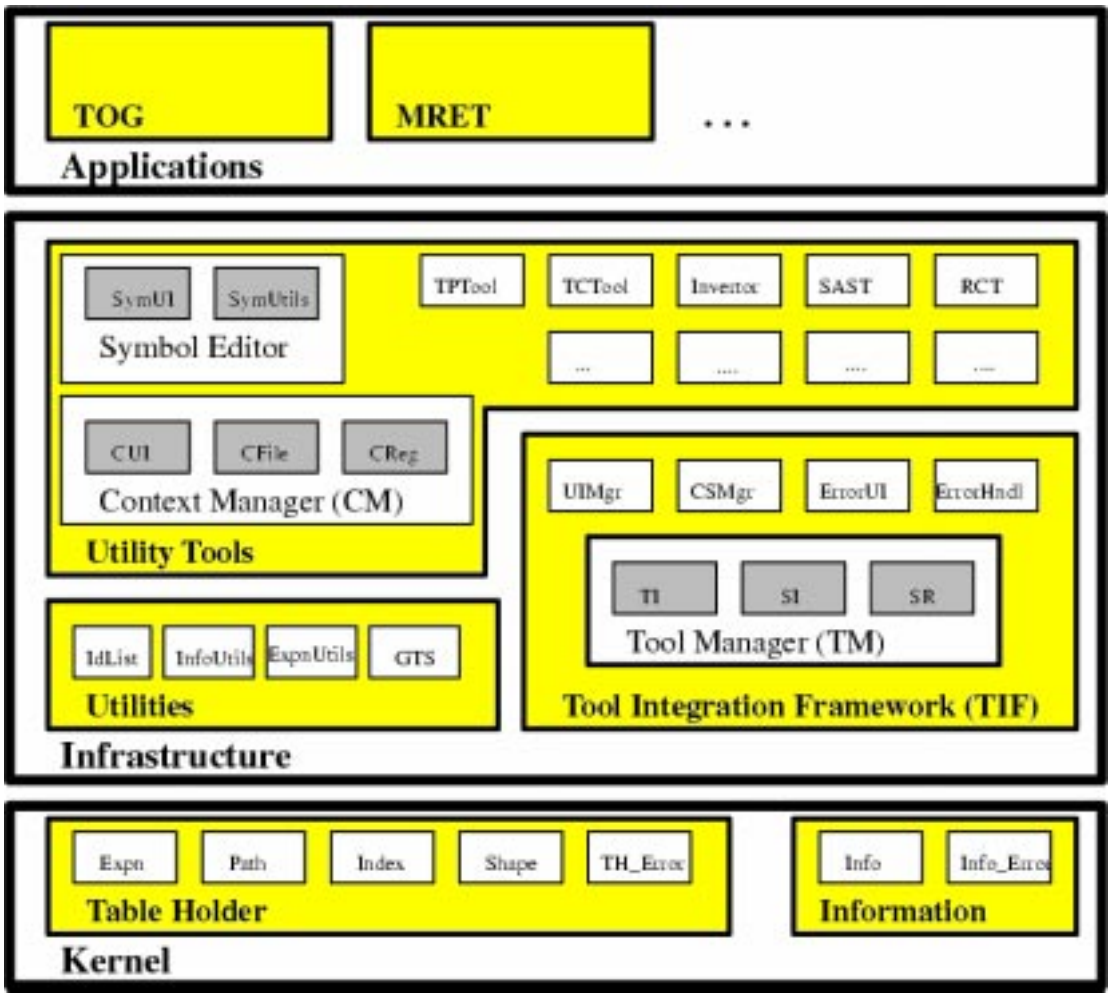


Figure 4: TTS Design Schematic

1.3.1 Tabular Expressions and Table Holder

TTS manipulates tabular expressions, which are descriptions of software elements. These expressions are stored in the Table Holder [13] in the form of a tree (see Figure 5).

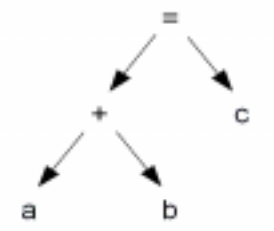


Figure 5: A syntax tree for expression a+b=c

Each node in a tree refers to a symbol and a set of child nodes. The symbol represents the relationship between the ordered child nodes. Thus, each node in the tree represents a root of a sub-expression, and the root node represents the entire expression. This tree can be conceptually thought of as a "syntax tree" – allowing for the interpretation of the precedence order of operations by simply examining the structure of the tree.

1.3.2 Symbols

For every symbol in the expression there is a set of associated data. Data stored in the Symbol Table [13] holds detailed information about the symbol, such as its name, type, arity, font family and font face to be used for display. In the case of a table symbol it contains additional information such as the table's cell connection graph (CCG) type [11], and the predicate and relation rules.

1.4 Need for a Tabular Expression Editor

Tabular expressions can be used as the basis for precise software documentation. To construct a tabular expression a table editor is needed. A table editor is one of the most important tools in TTS – without it one cannot easily create tabular expressions.

Once a tabular expression is created it can be manipulated by the other tools in TTS. Therefore, it is important that a table editor constructs correct expressions. It is also important that creating tabular expressions is easy so that the user can fully concentrate on documenting the software with little distraction.

1.5 Criteria for an Effective Tabular Expression Editor

Tabular expressions are similar to two types of objects: equations and tables. To understand how to make editing of tabular expressions easy and effective, we surveyed applications that can create and edit tables and equations.

The set of applications surveyed is limited to well-known text editors: Microsoft Word [17], Adobe FrameMaker (FM) [18], Interleaf (IL) [19] and their associated equation editors. Also surveyed is spreadsheet editor Microsoft Excel [20] and domain

specific tools such as Grand Table Interface (GTI) [21] and Table Construction Tool (TCT) in TTS. These applications are representative of the wider class of table and equation editors because they employ well established editing techniques that are most familiar to the average user.

By examining the editing techniques used in the surveyed applications we established that the following essential criteria must be met by an effective tabular expression editor:

1. **Correct expressions** – the tool must create correct tabular expressions.
2. **Verification and error analysis** – when loading an expression the tool must verify it. In the case where a loaded expression is incorrect, sufficient error analysis must be provided for the user to take steps correct it.
3. **Free input with flexible validation** – the user can type in or paste the text of the expression directly without constraints; syntactic validation can be done either while the user is typing or when explicitly specified by the user.
4. **Choice of preferred notation** – the expressions should be shown with the user's preferred notation.
5. **Expression state indicators** – the expressions should use indicators to represent the expression state. For example, using placeholders to indicate missing elements.
6. **Configurable table types** – new table types can be added easily without requiring any source code changes.
7. **Insertion and deletion of cells** – the tool must allow insertion and deletion of rows and columns of cells in a table, but only where it is permissible.
8. **Editing operations** – common editing operations such as cut, copy, and paste need to be supported.
9. **Allow for using and defining new symbols** – the user need to be free to use symbols that are not yet defined. The user also needs be able to define the new symbols while editing an expression.

10. **Tables in tables** – tables are valid expressions, so the user should be able to create tables in the cells of tables.
11. **Error correction** – simple syntax errors should be automatically corrected.
12. **Extensibility and the ability to configure** – users should be able to extend and configure the behaviour of the tool in order to better adjust its function to their needs.
13. **Expression helpers** – the editor should use expression helpers to clarify the structure and context of the expression being edited.

1.6 An Effective Tabular Expression Editor

Tabular expressions can be created using the existing tools. However, by contrasting the surveyed applications against the given criteria in Table 1 none of the existing applications meet the criteria, and therefore none are effective tabular expression editors.

	Criteria	Excel	Word	FM	IL	GTI	TCT
1	Correct expressions	No	No	No	No	No	Yes
2	Verification and error analysis	No	No	No	No	No	Yes
3	Free input with flexible validation	No	No	No	No	No	No
4	Choice of preferred notation	No	No	No	No	No	No
5	Expression state indicators	No	Yes[1]	Yes[2]	No	No	Yes
6	Configurable table types	No	No	No	No	Yes[3]	No
7	Insertion and deletion of cells	Yes	Yes	Yes	No	Yes	No
8	Editing operations	Yes	Yes	Yes	Yes	Yes[4]	No
9	Allow using and defining new symbols	No	No	No	No	No	No
10	Tables in tables	No	Yes	No	No	No	No
11	Error correction	No	No	No	No	No	No
12	Extensibility and ability to configure	No	No	Yes	No	No	No
13	Expression helpers	No	No	No	No	No	Partial[5]

Table 1: Characteristics of Surveyed Applications

It is possible to create an editor that meets all criteria. The Table Input Method (TIM), shown in Figure 6, is a new TTS tool that was constructed with the goal of

[1] Only when using the Microsoft Equation Editor.
[2] Only when using the FrameMaker Equation Editor.
[3] Dimensions only; GTI is limited to a maximum of five grids.
[4] Text only.

meeting all the criteria. The following describes how TIM meets each of the 13 given criteria:

1. TIM creates correct tabular expressions.
2. TIM validates loaded expressions, and warns the user when the loaded expression is invalid.
3. TIM allows expressions to be typed in freely, and lets the user choose when to validate the input.
4. TIM lets the user choose his or her preferred notation.
5. TIM uses placeholders to indicate missing sub-expressions.
6. TIM has configurable table types. Grid location and grid dimensions are loaded at run time from the configuration file.
7. TIM allows the user to correctly insert and delete rows and columns of cells in the tables.
8. TIM has all the standard table and cell editing operations. It even allows entire expressions to be pasted into a cell, or to create new expressions form the contents of a cell.
9. TIM can operate on undefined new symbols, and it allows the user to define new symbols from within, without external intervention.
10. TIM can create tables in tables.
11. TIM is capable of correcting syntax errors and indicating where the missing sub-expressions might be found.
12. TIM is easily extended through its input parser framework architecture, configurable table types, and extensible symbol class value ranges.
13. TIM can show the table type (CCG with predicate and relation expressions) and the syntax tree of the current expression.

[5] TCT shows sub-expressions by underlining them. However, for table's sub-expressions, the table's CCG type as well as the predicate and relation rules cannot be shown. This can be a source of confusion for the user.

Therefore, as shown by TIM, it is possible to create a tabular expression editor that fulfills all necessary criteria.

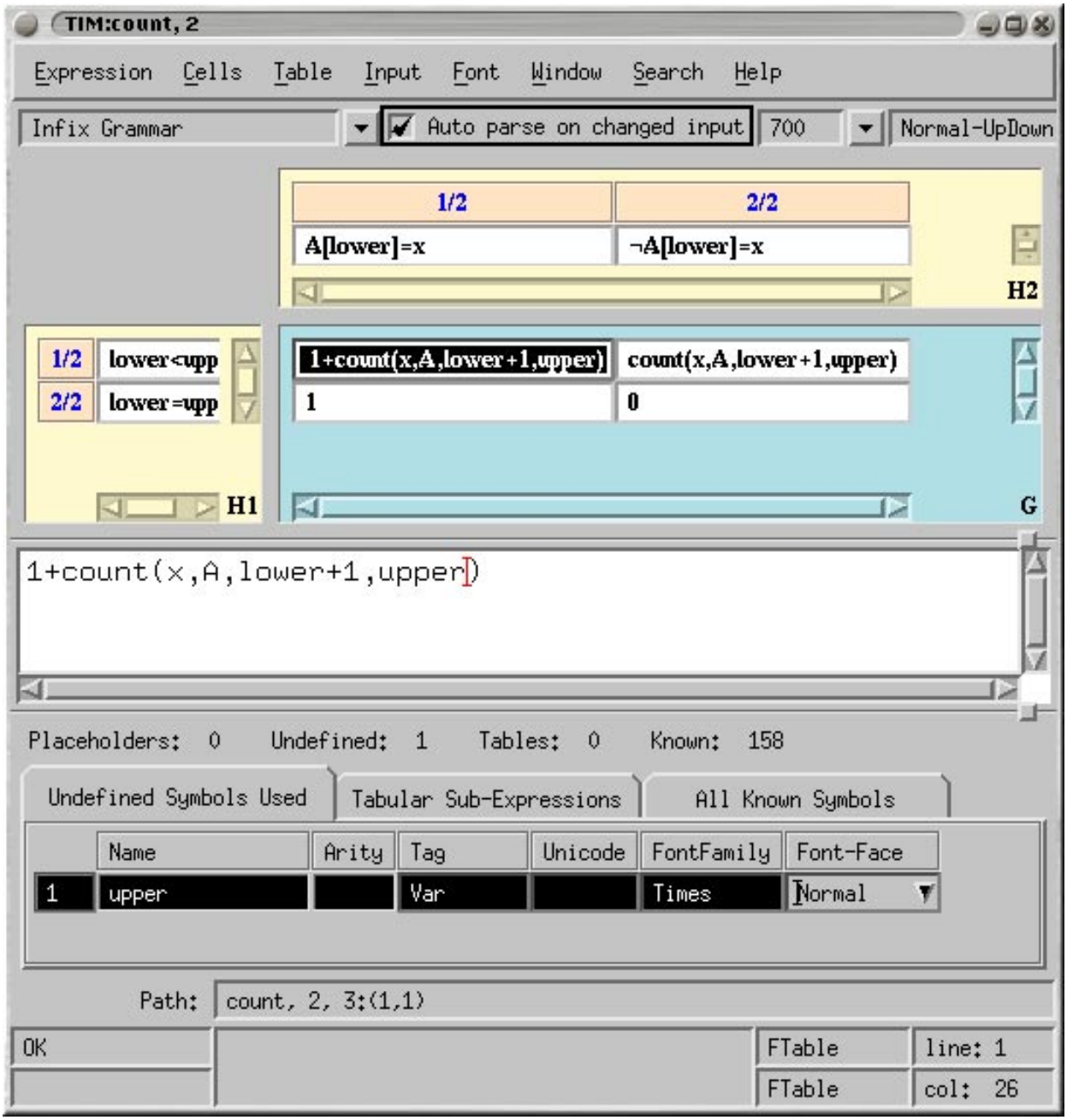


Figure 6: Constructing tabular expressions in TIM

1.7 The Importance of an Effective Table Editor

Precise software documentation method must be both mathematically sound and easy to use.

The application of tabular expressions in software documentation is mathematically sound and has been shown to increase the quality of the software product [1] [3]. In practice, however, the existing tools cannot create such tabular documentation easily. Uers can be quickly discouraged and unable to document more than the most trivial of examples. In order to make tabular notation practical and effective, the expression editors should meet all criteria outlined above.

2 Survey of Expression and Table-content Editing Applications

2.1 Introduction

TTS users should be able to manipulate tabular expressions in ways that are consistent with commonly used applications. Skills learned using these other applications should be easily transferred to solve similar problems in similar contexts arising in TTS.

Ideally, the editing tool would allow for the construction TTS expressions in an easy and intuitive way. Tables in TTS are already similar to two different classes of objects for which familiar editors already exists: equations and tables.

A tabular expression editor should leverage on the familiar concepts used in spreadsheets, text editors, and equation editors, and should apply them in the context of constructing tabular expressions.

In this survey we have selected well-known and commonly used applications that are capable of editing tables and equations. These are: Microsoft Excel, Microsoft Word, Adobe FrameMaker, Interleaf, Grant Table Interface [21], and Table Construction Tool. The editing techniques used in these applications are those most familiar to the average user. These are the techniques that allows for the easy creation of tabular expressions. We will now explore some of these applications in more detail.

2.2 Microsoft Excel

Microsoft Excel is a widely-used spreadsheet application that handles tabular information. Excel allows the user to manipulate data in tabular form. The program consists of a single regular grid with single row of column headers and a single column of row headers. In the cells, the user can enter either data or formulas. The relevant

features of Excel for the purpose of displaying and editing tabular information are the following:

1 The user can edit the contents of the cells unless the cells are restricted. Cells can display either the evaluated results of the formulas (default behaviour) or the formulas themselves (display style is toggled by typing CTRL+‘).

2 The formulas in the cells use infix notation. A wide range of functions is available to the user. These functions take a set of arguments and return a value.

3 Users are provided with visual clues to decipher complex formulas:

i. Parentheses control the order of infix operations. Function calls also use parentheses to delimit their arguments. The function calls can be nested. A formula can potentially use a very large number of parentheses. To help the user distinguish which set of parentheses belongs to which function call or operation, a matched pair of parentheses is shown in bold lettering when a cursor is near either parentheses of the pair.

ii. Arguments to functions can be references to other cells. To help locate the cells when a formula is being edited, each cell reference in the formula and the cell in the spreadsheet are shown in a unique matching colour.

4 The user can select, copy, cut, paste and clear a range of cells. When pasting, the user has the option of pasting either an exact copy of the formulas or only their values.

The limitations of Excel, with respect to tabular expressions are:

1. Excel does not handle general tables. It uses a specific table format, known as a “spreadsheet”.

2. Excel has its own spreadsheet context for its formulas – it does not handle the type of tabular expressions used in TTS. One can enter TTS style expressions into the cell (as text), but there is no verification of the expressions’ validity.

3. There are currently no translation tools that convert Excel file formats to TTS file formats, although it is possible to create such a tool.

4. Excel cannot hold tables within cells. There is no concept of a table as a valid expression that can exist within a cell.

Tables created in Excel are not TTS tables. Excel's purpose is different and the data format in Excel is also different from what TTS uses so that the Excel files cannot be used directly in TTS.

It might be possible to use Excel as an input device, and export the contents of a table in a comma or a tab delimited text format. This exported format can then be used as an input into a translation program to create a corresponding TTS file. This program does not exit right now and would have to be created. However, even if the translation program existed, it would not provide the user with immediate feedback about the correctness of the syntax of the expressions. Besides, representing inner tables would be a significant challenge to overcome.

2.3 Microsoft Word

Microsoft Word is perhaps the most popular text editor. In addition to allowing the user to edit text, it permits editing of equations and tables.

2.3.1 Microsoft Word Equation Editor

The information contained in tabular expressions are mathematical expressions or equations. Microsoft Word Equation Editor permits the user to insert equations into a document. The Editor provides the user with layout facilities to construct an equation, but it does not provide any lexical or syntactic checking.

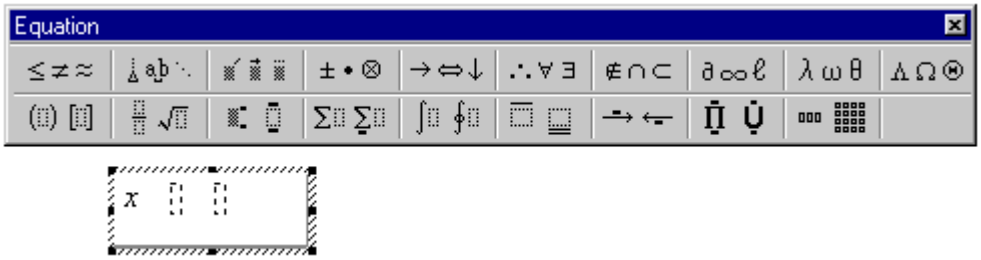


Figure 7: Word Equation Editor

To construct an equation, the user inserts an Equation Editor object into a document. Word immediately enters into object edit mode. When in edit mode, the user is presented with the equation entry area, new application menu items and a palette of symbol menus. Initially the entry area contains a single input "box" (▯) as a placeholder for entering the equation. The user can then insert various elements into the box by using the keyboard and palette menus. The elements can be text, simple symbols, complex symbols (symbols with extra boxes), and different types of boxes or box sets (Figure 7).

It should be noted that Word's Equation Editor does not impose any particular notation on the equation, and the user is free to enter any symbol into a placeholder. That is, the user can choose to construct an equation in prefix, infix or postfix notation. No syntactic information about the equation is retained.

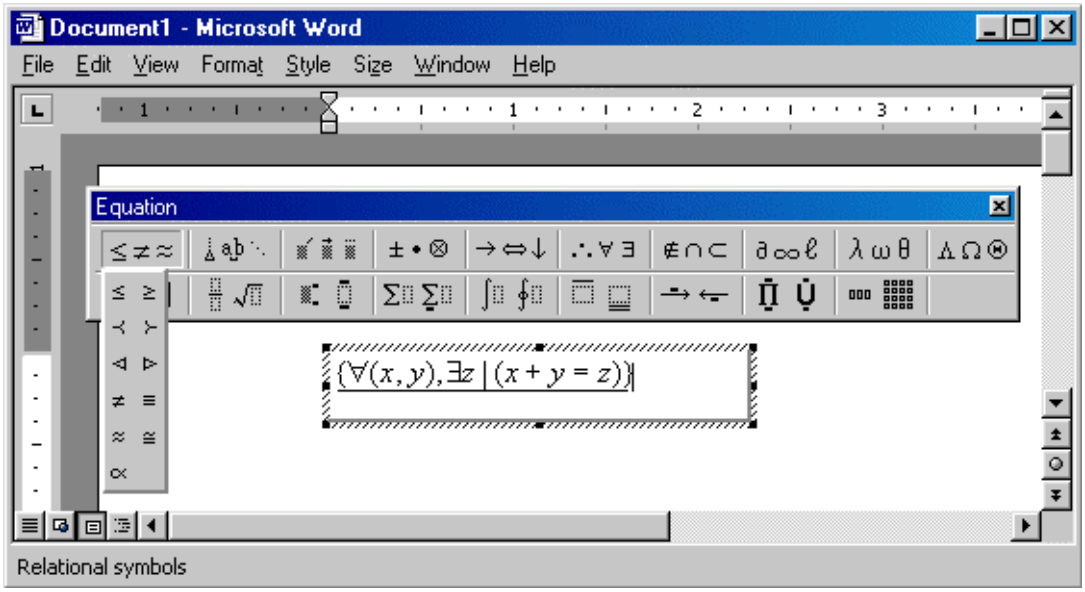


Figure 8: Word Equation Editor - completed equation

By successively adding various placeholder boxes and inserting various symbols, the user can construct an entire equation (Figure 8). The user can adjust the font size, spacing, and other aspects of the equation. Once the equation is completed, the user can close the equation palette. The equation is inserted as an object into the document. To change the equation, the user opens the object and edits it.

2.3.2 Microsoft Word Tables

Microsoft Word allows the user to create tables by selecting **Table→Insert→Table…** from the menu and specifying table parameters such as its dimensions, in the Create Table dialog.

Each cell in a table can contain any text, equation, or other table (see Figure 9). The user has complete control over formatting so that borders, background and foreground colours, text alignment, font and size for each cell can be modified.

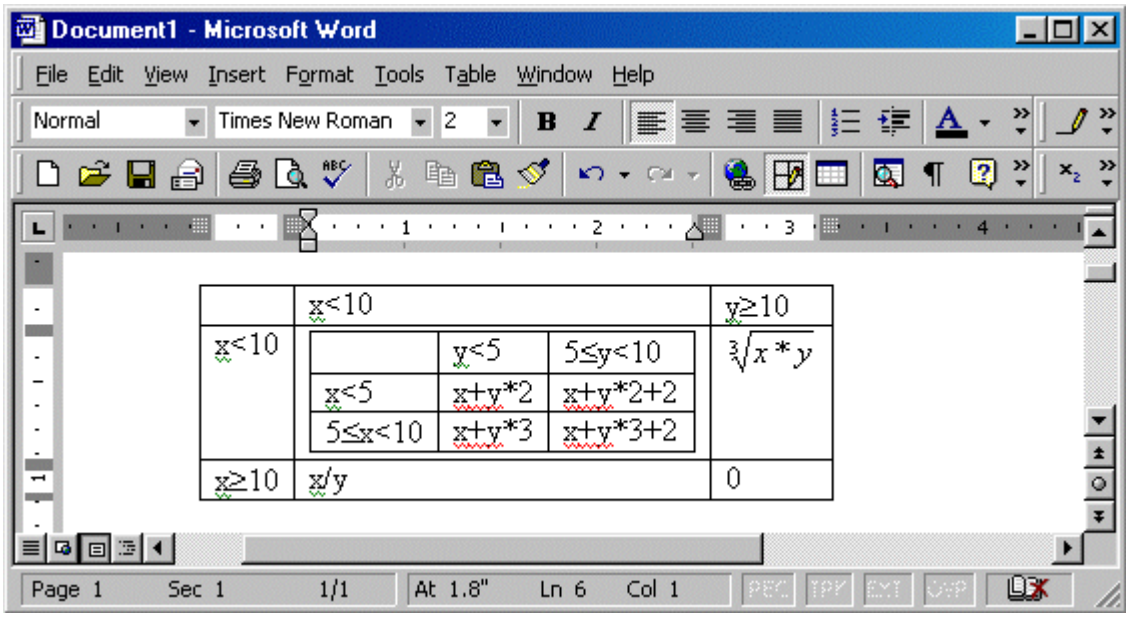


Figure 9: Word – a table in a table

Given the cell content and formatting flexibility, the tables can represent anything the user wants. However, there are no means of validating the contents of the table aside from performing a spell check on its textual content. There are also no means of relating

one cell to another, or of specifying formulas that depend on the contents of other cells, as in Excel. Tables in Word are purely for visual display.

It is possible to use Word as to create documents with tables that can be translated by an external program into tabular expressions suitable for TTS. In particular rich text format (RTF) documents lend themselves easily for further processing. However, this solution has several shortcomings:

1. The user does not have immediate feedback about the structure, state, or correctness of the expression being edited
2. The construction of tables containing inner tables can be challenging and error prone
3. The user cannot declare new symbols as they are needed

2.4 Adobe FrameMaker

Adobe FrameMaker is another popular word processing application that can create tables and expressions.

2.4.1 FrameMaker Equation Editor

To create an expression in a FrameMaker document the user must use the FrameMaker Equation Editor. In many respects the FrameMaker Equation Editor is similar to the Word Equation Editor, but there are important differences.

As in Word, a user wishing to add an equation would add an equation object into the body of the text. When editing the object, the user is presented with an equation entry area. Object symbols are displayed in a palette (Figure 10).

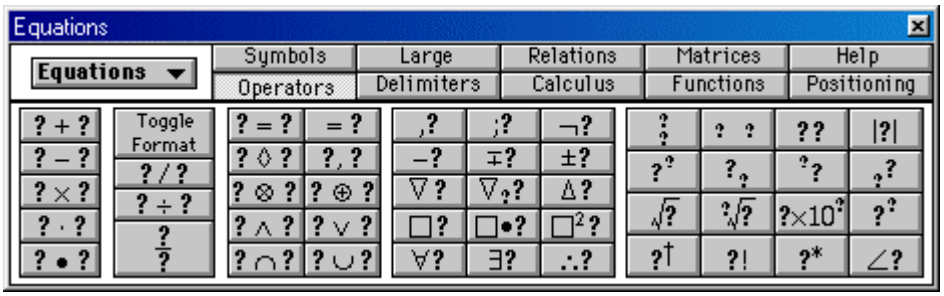


Figure 10: Equation palette in FrameMaker

Again as in Word, the user expands placeholders to contain more and more complex elements of the equation. However, the approach in FrameMaker is more structured than that in Word: all operations have a predefined number of terms, and all use infix notation explicitly. For example, to use addition, the user would have to input the addition structure into a placeholder containing the "+" symbol and two placeholders around it. In Word, a "+" symbol is simply inserted into an available placeholder.

The FrameMaker approach permits the application to retain syntactic information about the expression being constructed. One of the advantages is that users are more likely to construct a syntactically correct expression than when they use Word. For example, in Word it is possible to construct an equation that looks like this: "x + + y = z". It is not possible to construct such an expression in FrameMaker.

Also, the FrameMaker approach enables the user to mentally chunk the equation into more manageable smaller parts. This is particularly useful when the equation is complex. A user can focus on the element at hand, knowing that she/he will be reminded about any undefined elements later through the existence of unfilled placeholders.

Another advantage of the FrameMaker approach is that it permits the user to solve equations in place, because the internal structure maintained for the equation retains enough syntactic information to do so.

Another parallel to Word is that, once an equation is completed, the user closes the object that is then inserted into the body of the text (Figure 11).

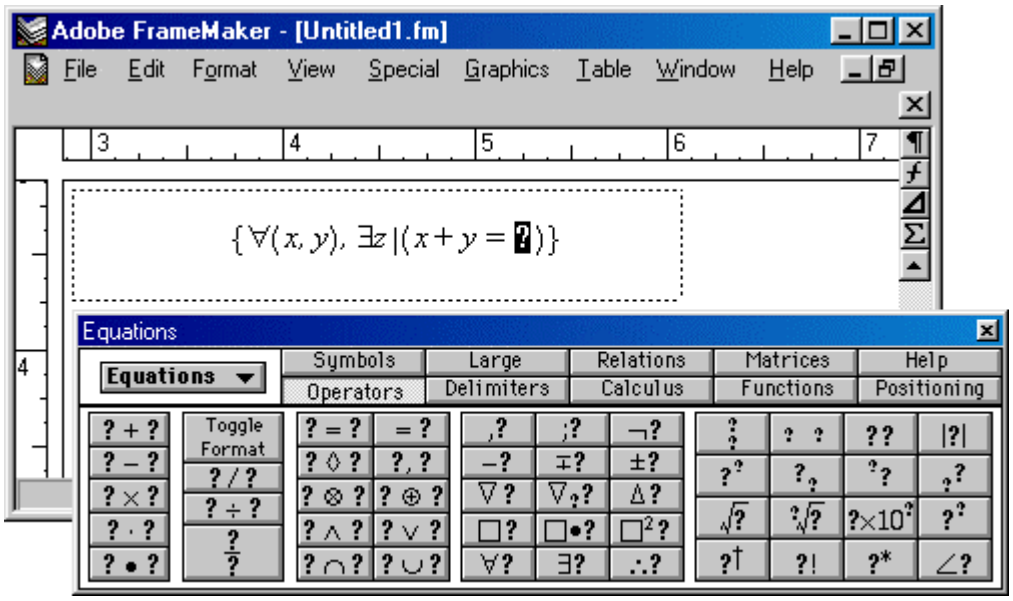


Figure 11: Editing equations in FrameMaker

2.4.2 FrameMaker Tables

In FrameMaker the user can create tables (see Figure 12). As in Word, the user has full control over the contents and the presentation format of the table. Unlike Word, FrameMaker cannot create a table within the cell of a table. Similar to Word, there is no syntactic verification of the table cell, and no formulas that can interrelate values in cells. Tables exist purely as a display device.

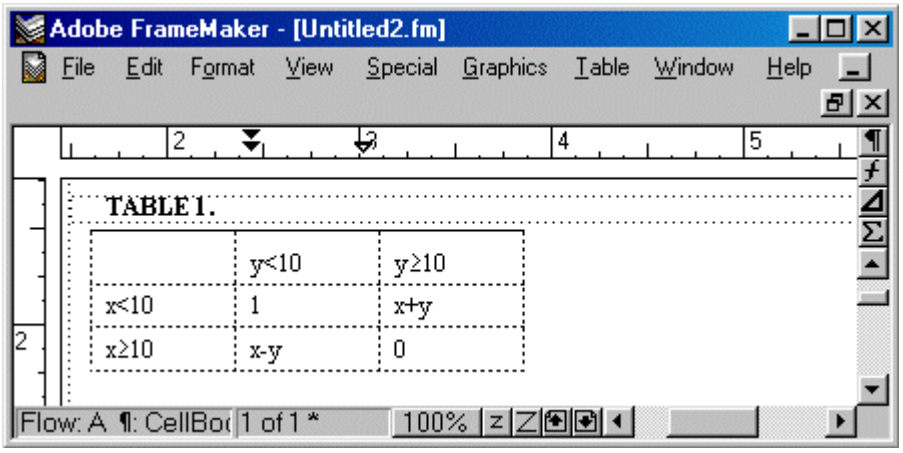


Figure 12: Creating tables in FrameMaker

2.5 Interleaf

Interleaf is a text editor that runs on Unix. This program provides basic text editing facilities, with support for equations and basic tables.

2.5.1 Interleaf Equation Editor

Interleaf Equation Editor (IEE) is an application that helps the user create equations within the Interleaf document. IEE takes a different approach to constructing equations than either Word or FrameMaker. To create an equation, the user opens the equation editor. The equation editor is composed of three primary elements: symbol and operator lists, an input text box, and the equation display panel (Figure 13).

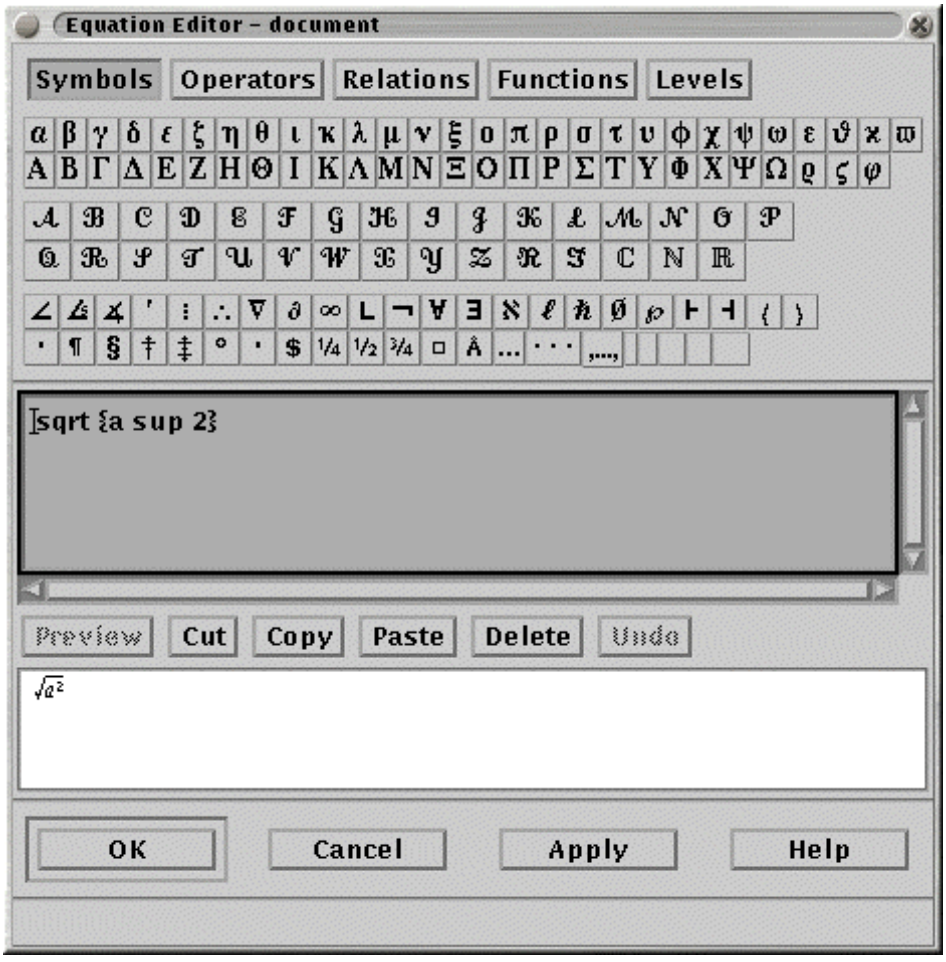


Figure 13: Equation Editor in Interleaf

The user types the equation in the input text box. If she/he wishes to enter a non-ASCII symbol, or operator, she locates and clicks on button displaying the symbol in the operator list. When the operator button is clicked, an operator keyword is inserted into the input text box. Once the user becomes familiar with the keywords, she may choose to simply type them in directly.

To display the equation in the display panel, the user must explicitly press the Apply button, since changes in the input text box are not immediately updated in the display panel. Once the user is satisfied with the equation, the user can click the OK button to place the equation shown in the display panel into the body of the text.

Unlike the FrameMaker Equation Editor, the IEE does not retain any syntactic information. It is similar to Word in that respect. The user is free to type in any expression, using any preferred notation.

2.5.2 Interleaf Table Editor

Interleaf has, perhaps, the simplest of table editing modes. Table cells cannot contain equations or inner tables. Cell borders (lines) also cannot be modified. Only text is permitted in table cells (see Figure 14).

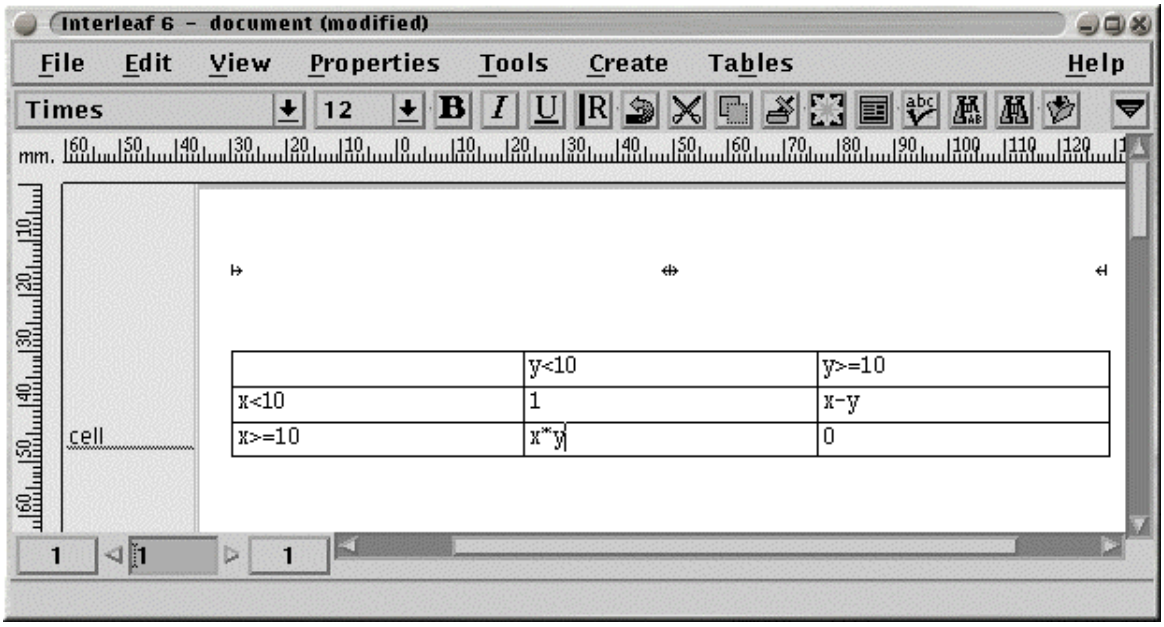


Figure 14: Editing a table in Interleaf

2.6 Grand Table Interface

The Grand Table Interface (GTI) is a tool that provides a user-friendly environment for constructing and modifying a wide range of tables. The user is presented with a general format to derive the desired table (see Figure 15).

The user specifies the dimensions of various grid elements of the table. For specific table types, some dimensions may be restricted. Once a table has been created, the user can enter Latin text in the cells of the table and then save those in a file (see Figure 16).

In GTI one can also insert and delete rows and columns from the table. GTI comes with tools to access and manipulate the GTI files. One of the tools is a translator that converts GTI files into files that can be loaded into TTS; GTI, therefore, provides an alternative to TCT for constructing tabular expressions.

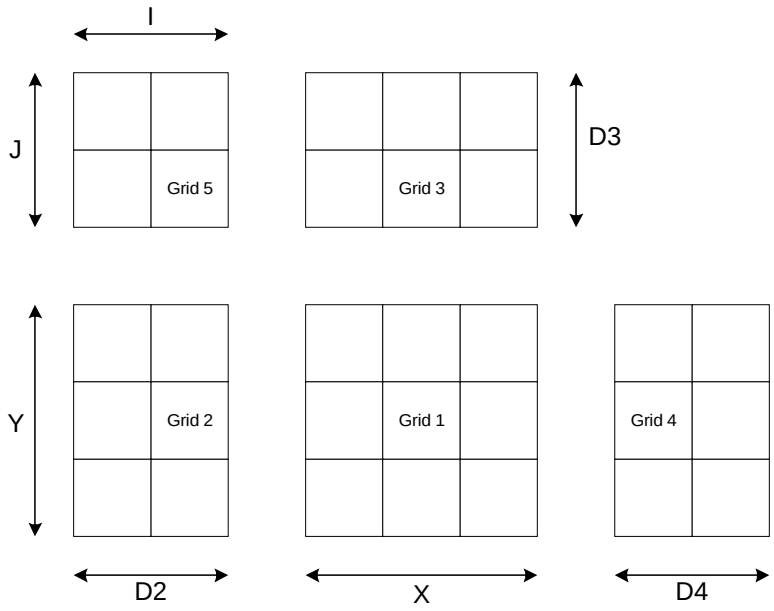


Figure 15: GTI – general tabular form

GTI provides intuitive operations to manipulate the contents of the table. Although GTI is an easy tool to use, it has some important limitations:

1. GTI allows any text to be entered into cells without restriction. There is no context in which cells are populated, no lexical or syntactic verification of the contents is performed, and no means of controlling expression input such as found in TCT are available. Thus, when using GTI to enter TTS expressions, the user may easily make a mistake that will not be noticed until the file created from the table's contents has been translated and loaded into TTS.

2. GTI cannot manipulate ranges of cells. Instead, it can only manipulate the table contents of individual cells. The user cannot select, cut, copy, paste or clear a range of cells. A user who wishes to reuse a range of cells in another table will have no choice but to first select a cell, then select its contents, then paste the contents into a cell in another table, and then repeat the process until all the desired cells have been copied.

3. Cell contents in GTI are limited to Latin text representation. This means that Symbol characters such as $\forall$, $\exists$, $\Rightarrow$, or $\in$ cannot be represented in GTI. However, TTS does use these symbols in expressions.

4. Cells cannot hold tables – a table in a table is not allowed.

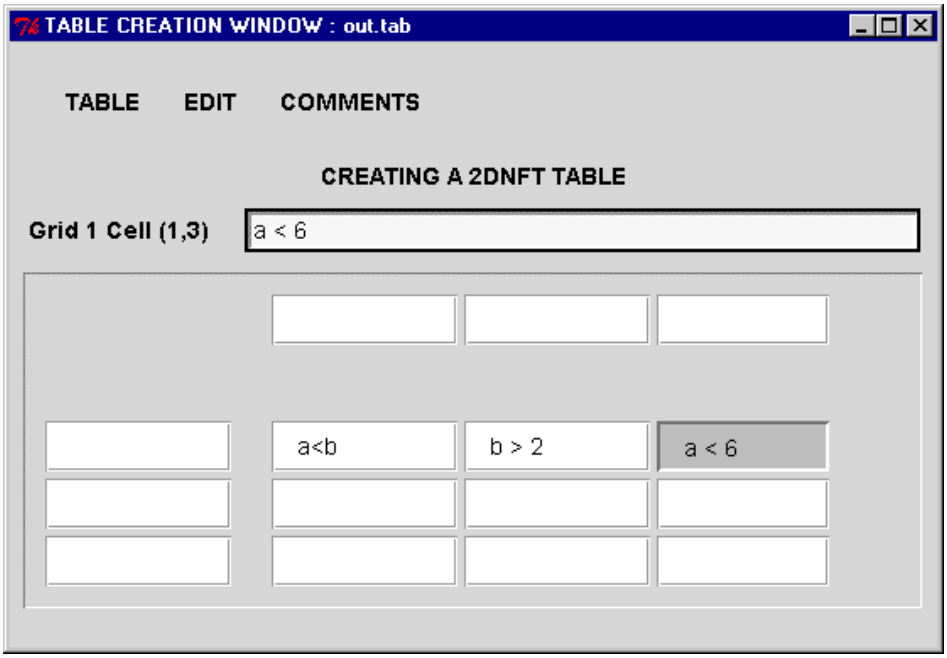


Figure 16: GTI Table Editor

2.7 Table Construction Tool

Table Construction Tool (TCT) is the existing TTS table editor. It allows the user to construct tabular expressions through a multi-window interface (see Figure 17). Expressions in TCT are shown using nested-prefix notation.

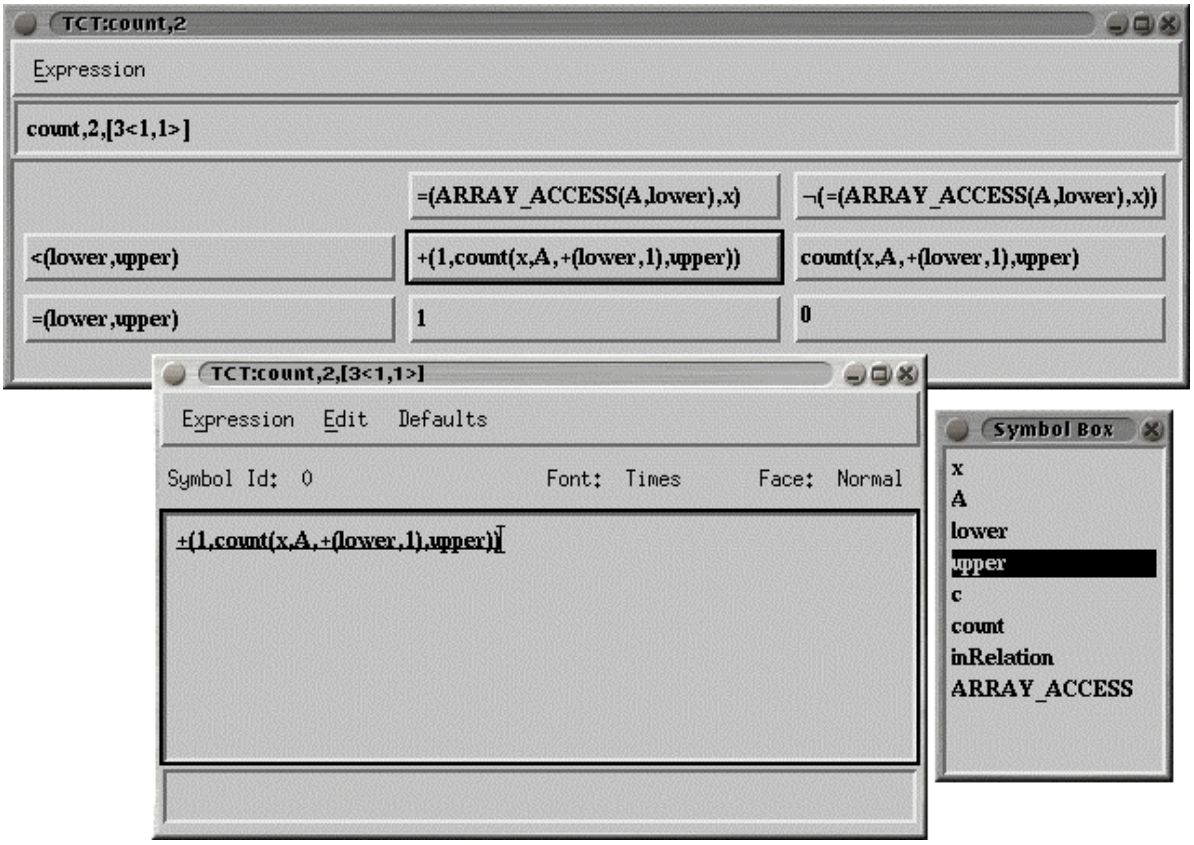


Figure 17: TCT - editing cell (1,1) in grid G (i.e. 3<1,1>)

The user constructs a table by first specifying its dimensions, and then by editing the individual cells. Each cell of a table can contain an expression. Tables in tables are not permitted.

Expression construction is tightly controlled by TCT. The user starts with a placeholder symbol "?", replaces each placeholder in the expression by inserting one of the default symbols from the **Defaults** menu, or by selecting a user-defined symbol from the **Symbols Box**. If the inserted symbol represents a function more placeholder symbols will be presented to the user, one for each function argument.

TCT is aware of the location where each sub-expression ends and begins. The user can only delete complete sub-expressions. A placeholder automatically replaces a deleted sub-expression.

3 Criteria for an Effective Tabular Expression Editor

Although tabular expressions in TTS can be arbitrarily large and complex, they must be correct and precise. There are different types of errors that the user can make in an expression: lexical, syntactic and semantic. By providing the user with a tool to construct these expressions, we can eliminate many common lexical and syntactic errors.

3.1 In addition to creating correct expressions, a useful tool should be intuitive and easy to use. When creating tabular expressions the application should re-use previously discovered and well thought out data-manipulation techniques. The user should not be burdened by the idiosyncrasies of the application. That is, the application should furnish the users with an environment that allows them to focus directly on the task of creating expressions as well as what these expressions mean.

The tool should accommodate a non-linear mode of thinking. It should let the user create any possible expressions, allowing the user to modify, expand and change parts of expressions as needed. The intent is to assist the user in creating the tabular expressions and not to get in the way. This is a very important point to understand, because it affects the way users perceive the value of the entire TTS. The inherent qualities of tabular expressions (validation, verification, simplicity, clarity, to name a few) and the tools that operate on them will not be fully recognized by the user unless it is easy to construct such expressions in the first place.

Having carefully examined applications that operate on entities similar to tabular expressions we can establish a set of criteria that must be fulfilled in a high-quality tabular expression construction tool. These criteria are as follows:

1. **Correct expressions.** All constructed expressions must be created and stored correctly.

2. **Verification and error analysis.** When an expression is loaded the tool must verify the syntactic correctness of the expression. In case of an error, sufficient information should be provided to help the user trace the cause of the problem. Most of the surveyed applications already do this.

3. **Free input with flexible validation.** The user should be able to type in the expressions freely, but the input should always be validated before changes are committed. Two distinct input validation modes must be supported: dynamic and static.

   When in the dynamic input mode, an expression is continually validated, thus providing the user with immediate feedback about the syntactic correctness of the expression. This is similar to the spell-as-you-type option in word processors.

   In the static input mode the expression is validated only after the user enters the entire expression and signals that she is done with the input (examples are Excel and Interleaf) – only then is the expression validated.

4. **Choice of preferred notation.** All expressions will be edited and shown in the user's preferred notation.

   Some of the tools surveyed enforce a specific type of infix notation (Excel and FrameMaker). The user has no choice in this matter, forcing the user to adjust to the tool. A more general approach would allow the user to select a preferred notation style. This would have a two-fold advantage:

   1) The user would manipulate expressions using the most familiar representation. Since this is configurable, another user can choose a different, more familiar representation. Thus, individuals can share their work, even though they "speak" a different notational "dialect".

   2) In order to facilitate expedience in creating new expressions, a notation should allow users to leave out bracketed rules. The difficulty is that TTS allows for many different operator symbols to be used, requiring the knowledge of operator precedence to correctly understand how the expression is structured.

This can be confusing to the user and is error prone – the ability to select a more restrictive grammar that uses delimiting symbols can be of great assistance to the user.

Note that providing a notational aid that clearly shows the operator precedence order for the current grammar, or a structural aid that explicitly shows the syntax tree of the expression should further enable the user to correctly understand the expression.

5. **Expression state indicators.** The validation process should provide the user with feedback about the expression and indicate what, if any, elements are missing by using visual indicators. [6]

6. **Configurable table types.** The user should only be permitted to construct tables with a valid TTS table type, shape and structure. However, the definitions of the table's structural parameters, such as locations of grids and grid placement, should be configurable so that new table types can be added easily without recompiling the application. This concept is well explored in GTI, and TIM uses it to good advantage.

7. **Insertion and deletion of cells.** Adding or deleting rows and columns in the table should be permitted. The newly created cells should be syntactically correct, for example, upon insertion the new cells can contain placeholders that the user can replace with complete expressions. This is a feature of Excel, where the new cells are empty (empty cells are syntactically correct in Excel). In GTI it is possible to add rows or columns, but there is no syntactic verification of the new structure.

8. **Editing operations.** A complete set of editing operations should be available and should work within an appropriate editing context. The editing contexts are: entire expressions, ranges of cells (see Figure 18), and the text of an expression. The mixing of contexts should also be allowed when it is logical to do so. For example,

[6] For example, FrameMaker leaves the "?" symbol as a placeholder for the portions of the expression that have not yet been completed. Microsoft Equation Editor uses an empty box (▯) as a placeholder.

it should be possible to apply copy and paste operations between an entire expression and a single tabular cell. The behaviour of the edition operation closely mimics those found in Excel, although Excel does not support the mixing of editing contexts (just like it does not allow tables in cells).

The editing operations are:

1. Clear – the selected items will be erased.

2. Copy – the selected items will be copied to the clipboard for temporary storage.

3. Paste – the contents of the clipboard will be copied to the current location.

4. Cut – the combined effect of performing the copy first followed by the clear operation.

Table A:

	$y < 0$	$y \geq 0$
x	By	$z + x$
y	$A(By) + B$	$A(z+x) + B$
z	$(xyz)(By)$	$(v - x)*(z+x)$

Table B:

	$y < -10$	$y \geq -10 \wedge y < 0$	$y \geq 0$
x			$z + x$
y			$A(z+x) + B$
z			$(v - x)*(z+x)$
r			

Figure 18: Copying and pasting range of cells from Table A to Table B

9. **Allow using and defining new symbols.** Undefined symbols can be used while an expression is being constructed. All undefined symbols used within the expression should be listed. The user will be prompted to define an undefined symbol and will have an option to defer the definition until the expression is committed.

10. **Tables in tables.** Tables are valid expressions, and therefore should be allowed to exist anywhere expressions are permitted. The application should allow the user to create a table within the cell of another table.

11. **Error correction.** A simple typographical error can make an expression invalid, and often they are difficult to spot. Whenever possible, simple syntax errors committed by the user should be corrected automatically.

12. **Extensibility and the ability to configure.** The tool itself should be a framework within which various components interoperate. These components can be replaced, or their behaviour adjusted. The user should be able to add new components or configure component behaviour so that the overall application is better tailored to the user's needs.

13. **Expression helpers.** The tool should facilitate the user's understating of the expression by showing the evaluation order, table types and grammatical rules.

Since each of these criteria complement each other, and directly influence the user's perception of the expression editor, all must be fulfilled to achieve maximum ease-of-use and expression-construction effectiveness.

4 Characteristics of the Surveyed Tools

4.1 Introduction

So far we have discussed how the surveyed software products have some characteristics that would be useful for the purpose of creating tabular expressions. However, none of the tools fulfill all the criteria to the extent that the user can create tabular expressions effectively, as summarized below.

Note that the focus here is only on the ability to create tabular expressions, not on the effectiveness for each tool to perform the task they were designed to do. That is, the effectiveness of FrameMaker or Microsoft Word as a text editor is not in question.

Finally, whenever the tool's characteristic is marked as "Free Style" this indicates that there is no built-in mechanism in the tool to fulfill the corresponding requirement - the onus is on the user to fulfill it manually, and is therefore a failure to meet the requirement.

4.2 Microsoft Excel

The similarities between spreadsheets and tabular expressions are superficial. Excel is not an effective tool for editing tabular expressions; the user can input text into the cells, but there is no expression validation or verification, and tables in cells are not allowed. Excel cannot be made aware of different types of tables and their structural and semantic information.

Microsoft Excel		
1	Correct expressions	No
2	Verification and error analysis	No
3	Free input with flexible validation	No – Free style[7]
4	Choice pf preferred notation	No – Free style
5	Expression state indicators	No
6	Configurable table types	No – Free style
7	Insertion and deletion of cells	Yes
8	Editing operations	Yes
9	Allow using and defining new symbols	No – Free style
10	Tables in tables	No
11	Error correction	No
12	Extensibility and ability to configure	No
13	Expression helpers	No

Table 2: Characteristics of Microsoft Excel

4.3 Microsoft Word

Microsoft Word is strictly a word processor. It allows for the creation of tables where each cell can contain anything, including other tables. It is not capable of validating the expressions created and cannot guarantee expression correctness. The program is not aware of different table types – construction of tables is strictly a text formatting operation.

Microsoft word		
1	Correct expressions	No
2	Verification and error analysis	No
3	Free input with flexible validation	No – Free style
4	Choice pf preferred notation	No – Free style
5	Expression state indicators	Only in Equation Editor
6	Configurable table types	No
7	Insertion and deletion of cells	Yes
8	Editing operations	Yes
9	Allow using and defining new symbols	No – Free style
10	Tables in tables	Yes
11	Error correction	No
12	Extensibility and ability to configure	No
13	Expression helpers	No

Table 3: Characteristics of Microsoft Word

[7] Although the user can type in an expression, there is no validation at all, and therefore this criterion is not met.

4.4 FrameMaker

The characteristics for FrameMaker are similar to those of Word, except that FrameMaker does not support tables in table cells. Like Word, it is strictly a word processing application, and tables exist purely for organizing and formatting text.

FrameMaker		
1	Correct expressions	No
2	Verification and error analysis	No
3	Free input with flexible validation	No – Free style
4	Choice pf preferred notation	No – Free style
5	Expression state indicators	Only in Equation Editor
6	Configurable table types	No
7	Insertion and deletion of cells	Yes
8	Editing operations	Yes
9	Allow using and defining new symbols	No – Free style
10	Tables in tables	No
11	Error correction	No
12	Extensibility and ability to configure	No
13	Expression helpers	No

Table 4: Characteristics of FrameMaker

4.5 Interleaf

Interleaf is the least sophisticated word processor in this evaluation. Table cells can only contain text and expressions are not permitted. Like other word processors, it can only edit text – no verification or validation of tabular expressions can be done.

Interleaf		
1	Correct expressions	No
2	Verification and error analysis	No
3	Free input with flexible validation	No – Free style, only text in cells
4	Choice pf preferred notation	No – Free style
5	Expression state indicators	No – Free style
6	Configurable table types	No
7	Insertion and deletion of cells	No
8	Editing operations	Yes
9	Allow using and defining new symbols	No – Free style
10	Tables in tables	No
11	Error correction	No
12	Extensibility and ability to configure	No
13	Expression helpers	No

Table 5: Characteristics of Interleaf

4.6 Grand Table Interface

GTI, on its own, is not an effective tool for editing tabular expressions. It lacks validation and verification of the input, it must rely on external tools to verify expression correctness, and it does not support the table in table requirement.

GTI		
1	Correct expressions	No
2	Verification and error analysis	No
3	Free input with flexible validation	No – Free style
4	Choice pf preferred notation	No – Free style
5	Expression state indicators	No
6	Configurable table types	Yes, five grid structure only
7	Insertion and deletion of cells	Yes
8	Editing operations	Text only
9	Allow using and defining new symbols	No – Free style
10	Tables in tables	No
11	Error correction	No
12	Extensibility and ability to configure	Yes
13	Expression helpers	No

Table 6: Characteristics of GTI

4.7 Table Construction Tool

TCT is the existing tabular editor in TTS. As such it deserves special attention - the characteristics and the analysis of TCT are presented in section 5 below.

5 TCT – Existing Tabular Expression Editor

We can now evaluate the existing expression editor, the Table Construction Tool (TCT), against the established criteria. TCT is one of the basic tools in TTS (see Figure 20). The characteristics of TCT are summarised in Table 7.

TCT		
1	Correct expressions	Yes
2	Verification and error analysis	Yes
3	Free input with flexible validation	No
4	Choice pf preferred notation	No
5	Expression state indicators	Yes
6	Configurable table types	No
7	Insertion and deletion of cells	No
8	Editing operations	No
9	Allow using and defining new symbols	No
10	Tables in tables	No
11	Error correction	No
12	Extensibility and ability to configure	No
13	Expression helpers	Partial

Table 7: Characteristics of TCT

TCT permits the creation of tabular expressions in the form of tables. However, the combined effects of the presentation method, the expression construction method, and many missing operations that are intuitively expected by the user, make the construction of tabular expression in TCT a considerable challenge, especially for novice users.

5.1 Notation

Infix is the most commonly used notation for the purpose of defining expressions in the sciences. In infix notation, operator precedence is necessary to decipher the expression evaluation sequence. The challenge, however, is that TTS defines and uses a large number of operators and, therefore, the infix notation would require the user to learn a long operator precedence list in order to understand how to correctly read an expression.

TCT shows the expression using nested-prefix notation. In nested-prefix notation, an expression consists of a symbol followed by a parenthetical list of expressions. By using the nested-prefix notation, TCT sidesteps the operator precedence problem. The user does not have to know the precedence of operators to determine in what order the sub-expressions are to be evaluated; operator precedence is not needed since the innermost sub-expressions (as delimited by a pair of parentheses) are evaluated first.

Despite the problem of the operator precedence, the infix is the most commonly known and used notation. It lends itself readily to the left-to-right way of thinking about expressions – it is a shorthand notation that replaces wordy, textual descriptions (see Table 8). Compared to the nested-prefix notation, the infix notation is more natural for most users.

Notation Type	Expression
English Text	There exist x and y, where x and y are element of P, such that x plus 1 implies y
Infix	$\{\exists x,y \mid x + 1 \rightarrow y; x, y \in P\}$
Nested-Prefix	$\mid(\exists(x,y),\rightarrow (+(x,1),y))$

Table 8: Examples of expression notation

TTS does not force any kind of notation at all. Expressions are stored in the form of a syntax tree that can be manipulated directly. The choice of the presentation is entirely left to the implementation. In fact, Table Printing Tool, used for formatting and printing the tables, uses the infix notation, presumably for ease of presentation.

Since the infix notation is more commonly known, it would be easier for the novice user to learn TTS if an infix notation is used in the input method, despite the infix notation's apparent shortcomings (complex rules of precedence and evaluation order).

Unfortunately, the user cannot choose the notation in TCT – the nested-prefix notation is hard-coded, and no expression comprehension-enhancing aids are used. Consequently, TCT fails the **preferred notation** requirement. It should be noted that extending TCT to support different notations and expression helper tools is not a viable option as it would be tantamount to rewriting the entire application.

5.2 Understanding Expression Evaluation

Any single notation is insufficient to allow the user to interpret an expression easily. The user should have, at her/his disposal, an array of aids to help her visualize and better understand the expression.

The basic premise behind the tabular notation is to allow the user to clarify complex mathematical expressions. Even at the cell level, the quest to simplify and clarify the expression should be pursued.

TCT does have the means to help a user find out how the expression is evaluated through sub-expression underscoring. The user can click on an operator or function in the expression, and the appropriate corresponding sub-expression will be underlined (Figure 19).

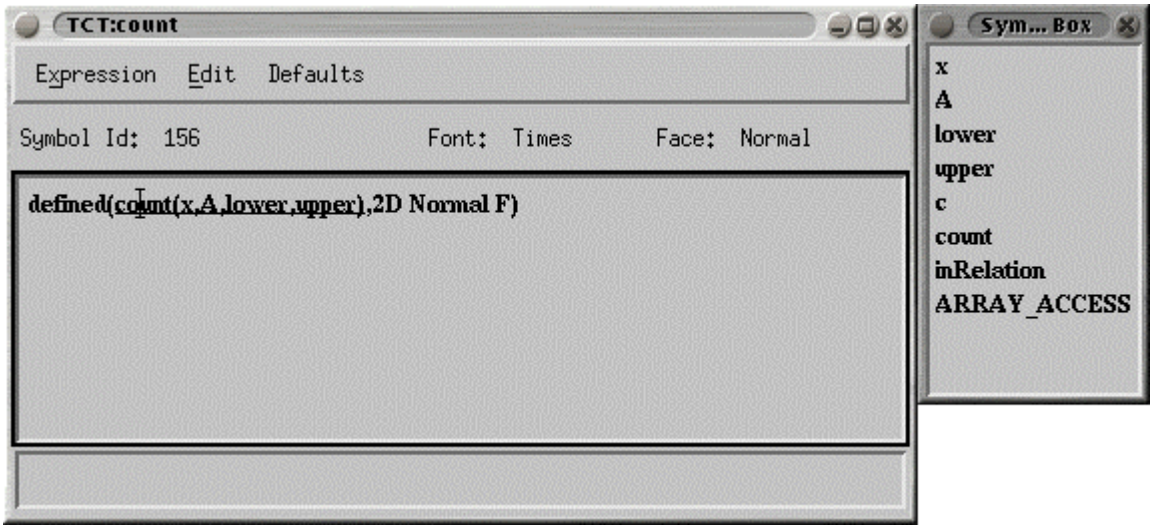


Figure 19: TCT notational aid – underscored sub-expression

Unfortunately, this does not work very well for lengthy expressions. Long expressions consist of many sub expressions, and the underscoring indicates only the current sub-expression of interest. The user is still forced to build a mental map of how each part of expression relates to the other parts, and this can be difficult for long and complex expressions.

One important expression helper is missing: when editing tables, the table type, predicate and relation rules are not shown. As a result, the user can quickly forget what kind of table is being edited.

Finally, while editing tables there are no clear indicators of where each table grid is located. The user must guess which cell belongs to which grid.

5.3 Expression Construction

In TCT, the expression is constructed starting from a single "?" placeholder symbol. A placeholder can be replaced by a constant, a function or a table symbol. Function symbols with an arity of at least 1 provide the user with further placeholders in place of undefined function arguments. By inserting a table symbol, a table can be defined. The cells in a newly defined table also initially hold placeholders (table construction is described in the next sub-section).

The expression construction consists of "growing" each placeholder into a fully qualified sub-expression, while the placeholder serves as the **expression state indicator**, indicating what else needs to be grown. There is no other method of creating expressions in TCT.

One advantage of the TCT is that there is no need for any sort of automatic syntax error correction. Controlled expression growth prevents the user from making syntactical mistakes. This advantage, however, is outweighed by some more serious shortcomings: it is difficult to create and modify expressions without first having a very clear mental image of where each of the expression's constituent components is to be located.

For example, to add a missing term, the user is often forced to reduce the existing expression to a point where it is possible to re-grow it into the newly modified form - adding even a simple missing additive term can be a trying exercise (see Table 9).

Expression State	Operations
*(a,b)	Position cursor at symbol “*”.
+(a,b)	Select “+” operator from Defaults→Functions→+ menu.
+(a,b)	Position cursor at symbol “a” and delete it – it will be replaced with symbol ”?”/
+(?,b)	Position cursor at symbol “b” and delete it – it will be replaced with symbol ”?”.
+(?,?)	Position cursor at first “?” and select “*” operator from Defaults→Functions→* menu.
+(*(?,?),?)	Position cursor at second “?” of the “+” expression and select “*” operator from Defaults→Functions→* menu.
+(*(?,?),*(?,?))	Position cursor at first “?” inside the first “*” sub-expression and replace it with symbol “a” by double-clicking on the “a” row in the Symbol Box.
+(*(a,?),*(?,?))	Position cursor at second “?” inside the first “*” sub-expression and replace it with symbol “b” by double-clicking on the “b” row in the Symbol Box.
+(*(a,b),*(?,?))	Position cursor at first “?” inside the second “*” sub-expression and replace it with symbol “c” by double-clicking on the “c” row in the Symbol Box.
+(*(a,b),*(c,?))	Position cursor at second “?” inside the second “*” sub-expression and replace it with symbol “d” by double-clicking on the “d” row in the Symbol Box.
+(*(a,b),*(c,d))	Done.

Table 9: Changing expression a*b into a*b+c*d in TCT

Even the insertion of symbols into an expression is very cumbersome: symbols can only be inserted by selecting an appropriate a menu item (for hard-coded default symbols), or an entry in user-defined list of symbols in the Symbol Box (see Figure 20). The expression cannot be typed-in by using the keyboard directly, even when one wishes to enter basic symbols such as +, -, =, 1, 2, etc.

Another obstacle in expression construction is that the user cannot create new symbols while editing an expression – all the symbols must be predefined before the expression is edited in TCT. This dramatically reduces user options – as the expression is

edited new concepts come to light requiring new symbols to describe them. The only method of adding the new symbols is to cancel the edit (the intermediate incomplete expressions cannot be saved), add the missing symbols and restart TCT to edit the expression. Consequently, it is clear that TCT fails the **free input with flexible validation** requirement.

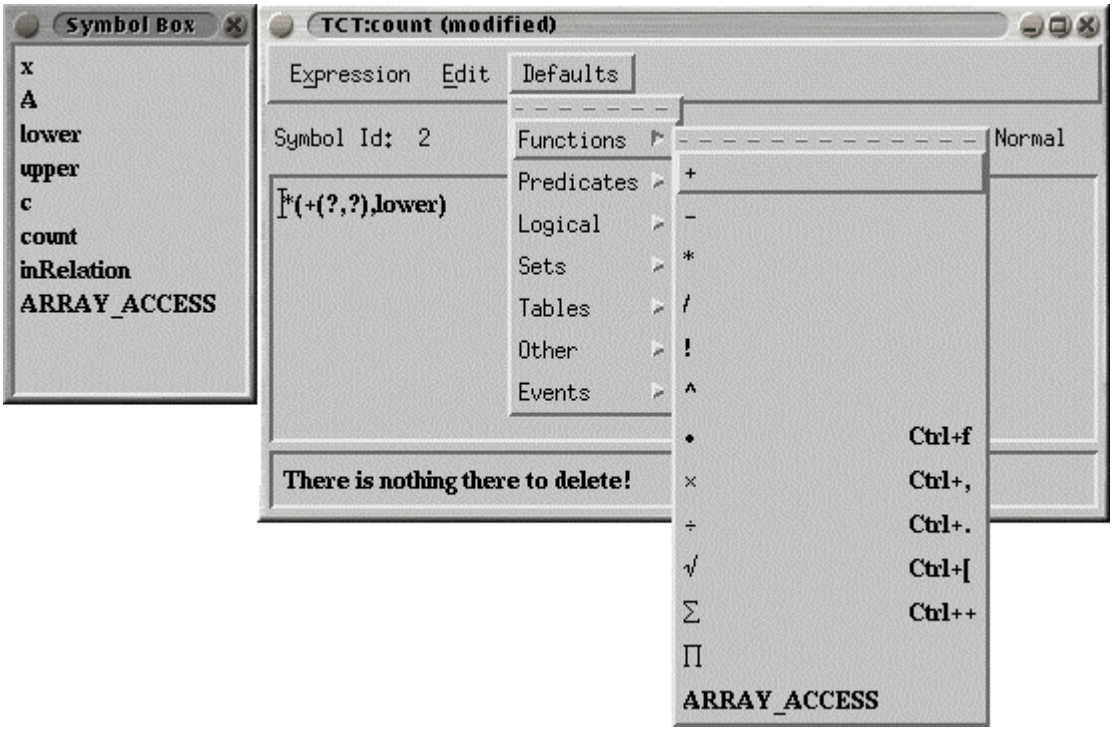


Figure 20: Entering sub-expressions in TCT

Moreover, the expression construction is further aggravated by a lack of **editing operations** – it is impossible to replace a placeholder with a copy of some other expression; to copy an expression, the user must recreate it. The table editor in TCT also does not have the editing operations – the user cannot cut, copy or paste cells.

5.3.1 Tables

Replacing a placeholder with a table symbol creates a table. In TTS, each table symbol defines the structure of the table: it declares the number of grids, interpretation rules, and the table type. In TCT the number of supported table types is **hard-coded**; to recognize a new table type, TCT has to be modified programmatically.

TTS does not explicitly specify the location and orientation of the table grids – these are left for interpretation by the tool. In TCT, for each table type, the number of grids and the location and orientation of grids is also preset.

The symbol does not declare the table dimensions – these are defined at table creation time. Selecting the table dimension is a crucial step as the **number of rows and columns cannot be altered later**. The only way to delete or insert a missing row or column is to reconstruct the entire table, one expression at a time.

The newly defined table is shown in a tabular GUI component, where each cell can be edited individually. Initially, each cell in the newly defined table contains a placeholder that can be grown into a fully defined sub-expression.

Tables in TCT **cannot hold other tables as their cells directly**: a cell must hold a non-tabular expression. Tables can hold other tables indirectly: the cells can hold an expression with a table symbol and editing the table symbol allows the user to edit its table.

Another shortcoming of TCT is that the user has no means to explicitly show the table predicate rule, relation predicate rule, and the cell connection graph (CCG) class. This is an important piece of information without which it is difficult for the user to interpret the table.

5.4 Correct Expressions, Verification and Error Analysis

By precisely controlling how each sub-element of the expression is being added, TCT ensures that the expression being constructed is always syntactically correct, and, consequently, that it fulfills the **correct expression** requirement. Incidentally, TCT also validates the expressions when they are being loaded from the file. Thus, TCT achieves the **verification and error analysis** objective.

5.5 Lack of Extensibility

TCT provides no means for tool extension. For example, the tool is hard coded for the types of tables it can operate on. Since the tabular expression is an area of

research that is continuously being developed, it is likely that new table types will be needed. In that situation TCT risks becoming obsolete.

TCT assumes a pre-defined set of default symbols that are accessible through the Defaults menu. The moment some of the default symbols are added, deleted or modified, the menus will no longer be completely valid – in fact they may be very misleading.

TCT uses only one type of notation – a notation that is not very common and which cannot be changed. A new notation cannot be added.

The tool does not allow the user to define and use keyboard mnemonics to insert functions and symbols. Symbol input is done through the hard-coded menu or the Symbol Box.

6 TIM – an Effective Tabular Expression Editor

In order to have an effective tool for the construction of tabular expression, entering the tabular expressions must be as simple and intuitive as possible. Table Input Method (TIM) (Figure 21) fulfills all the criteria for a good tabular expression editing tool as outlined in section 3 above.

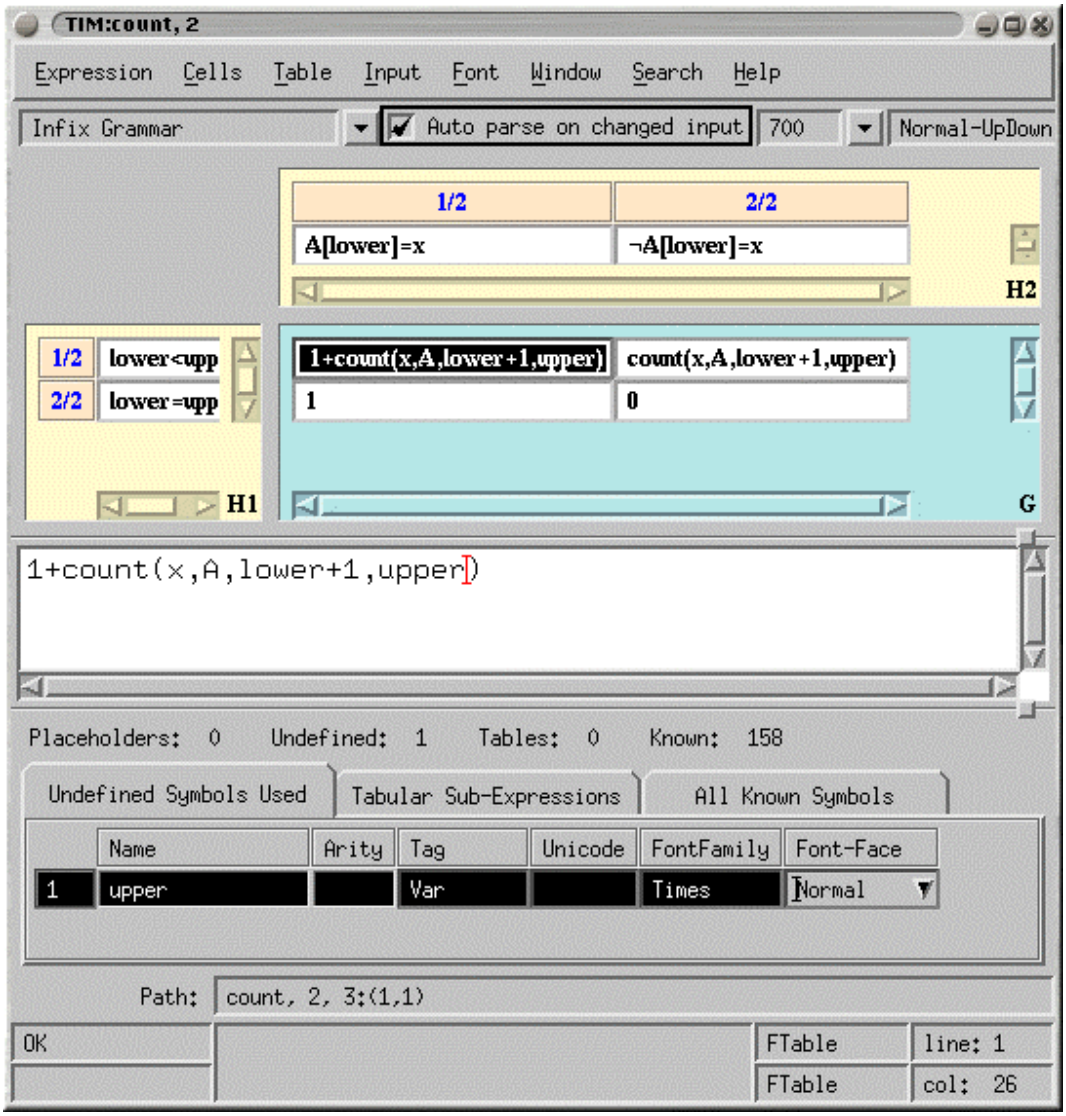


Figure 21: Table Input Method - editing tables

The characteristics of TIM are summarised in Table 10. Details on how each of the criteria is fulfilled are described in the following subsections.

TIM		
1	Correct expression	Yes
2	Verification and error analysis	Yes
3	Free input with flexible validation	Yes
4	Choice pf preferred notation	Yes
5	Expression state indicators	Yes
6	Configurable table types	Yes
7	Insertion and deletion of cells	Yes
8	Editing operations	Yes
9	Allow using and defining new symbols	Yes
10	Tables in tables	Yes
11	Error correction	Yes
12	Extensibility and ability to configure	Yes
13	Expression helpers	Yes

Table 10: Characteristics of TIM

6.1 Correct Expressions in TIM

Correctness is guaranteed through two co-dependent mechanisms: the table template and the input parser. The template guarantees that the table shape will adhere to the specified constraints, and the input parser validates the input converts the textual representation into an internally stored syntax tree. As long as both mechanisms are correct, the resulting expression will also be correct. Creators of templates and parsers should be very careful when making additions or changes.

TIM does not check for completeness - other tools in TTS do this job. TIM's purpose is to allow the user to input the contents of a tabled. Verification of correctness is limited to the creation of expressions with correct syntax trees. TIM guarantees only that the constructed expressions can be used elsewhere in the TTS system.

Also, defining the symbol semantics is also not a job TTS. Semantics are specified as part of the symbols, externally to the TIM. TIM is capable of displaying the semantics of the tables. The table semantic rules can be shown for the benefit of the user an informational aid about the type of the table being used. Semantics in TIM are used for no other purpose.

6.2 Verification and Error Analysis in TIM

TIM verifies each newly loaded expression. If the expression is not correct, the user will be informed of the error. Further details about the error can be obtained from the TTS log when an appropriate logging level is set.

6.3 Free Input with Flexible Validation and Automatic Corrections in TIM

The users of TIM can type in the expressions directly by typing in the text of the expression with all the characters and the symbols it contains. Then the entire text of the typed-in expression is parsed and validated for syntactic correctness governed by the currently selected grammar.

Validation can be either dynamic or static. In the dynamic mode the text of the expression is validated constantly as being typed in. The user can control when the input validation will be triggered by adjusting the length of the delay between the last keystroke and the start of the parsing process. The effect of this is such that the validation does not occur until there is a pause in the text input. When the pause is long enough the input is validated. If the input has not been changed, validation will not be triggered.

In the static mode the user activates the validation of the text of the expression explicitly, whenever the user deems it necessary – this allows the user to type in the entire expression unperturbed, without validation being triggered by the typing pauses. The expression validation will happen at the explicit request by the user, when the sub-expression window is closed, or when the expression changes are being saved.

6.4 Choice of Preferred Notation in TIM

In order to facilitate the user's choice of notational representation, TIM supports the ability to define an input-validating grammar. This grammar is used to derive a parser that is responsible for validating the user's input and converting it to an internally stored abstract syntax tree (AST) representation of the typed expression.

To display a stored expression in human readable form, the editor creates a string representation of the stored AST. To accomplish this TIM queries the parser for the grammar rules that match the nodes of the AST, and reconstructs the input string as the correct rules are being found. Thus, the string representation of the expression is totally dependent on the choice of grammar.

TIM comes equipped with a set of tools that make the creation of the parser very simple. All that is required is a new grammar definition file which is then compiled into a set of Java classes that implement the parser. The location of the parser is declared in the TIM configuration file. TIM loads the parser at start-up time. Note that in order to add a new parser to TIM, TTS does **not** need to be recompiled.

This flexible method of creating tailored parsers has many advantages. Chief among these are:

1. **Ease of input and ease of inspection.** These two objectives, seemingly opposed to each other, are both supported: the user can choose to use a less restrictive grammar (for example, one that makes the precedence rules implicit) to input an expression; to inspect an expression the user can choose a more restrictive grammar (one that makes the precedence rules explicit) that helps the user to determine that the expression is intended.

2. **Ease of collaboration.** Two users working with two different expression notations to collaborate on the same expressions without forcing either one to learn a new notation. They work with the notation they are most familiar with, independently of what the collaborator is using, provided they have a parser for the preferred notation.

To define a new parser, the user must first create a grammar definition file and then compile it into a parser (see Appendix B: Writing a Custom TTS Parser below). The parser is used both for parsing the input, and for recreating the display string from the internally stored syntax tree. This is done when an expression is loaded into editor, or when the user changes the parser (see Appendix A: A.5.4.1 Input Validation section below).

6.5 Expression State Indicators in TIM

TIM uses a placeholder, the "?" symbol, to indicate which part of the expression is yet to be specified. Placeholder symbols are injected into the input stream at the point where the parser determines that there is a missing sub-expression, according to the currently selected grammar rules.

A placeholder allows the user to quickly scan the input string and locate the expression that is not yet completed, and focus her/his attention on that part. Expressions that are syntactically valid but contain placeholders can be stored in TTS, since placeholders are valid symbols.

6.6 Configurable Table Types in TIM

Configuration files contain the table template properties: number of grids, location of grids, dimensions of grids, min and max number of rows and columns, etc. Grid themselves are placed in a master grid. The dimensions are controlled by specifications for each row of grids, the number of rows each grid will have, and for each column of grids how many columns each grid will have. Using this method we can ensure that the neighbouring grids have matching numbers of rows and columns.

For any given dimension, you can specify the min and max values, with valid numbers between 0 and a build-in value of MAX (currently maximum integer value). This way, for example, you can declare in the table template that the last grid-column has 0 minimum number of columns. At table creation time the user may choose to declare the last column of grids to be superfluous when she indicates the dimension to be indeed 0.

6.7 Insertion and Deletion of Cells in TIM

The user can indicate whether to delete the current row or column (the minimum allowed number or rows or columns is preserved) by selecting the appropriate action from the Table menu. The user can also insert a row (or column) below or above (left or right) of the current one (see Figure 22).

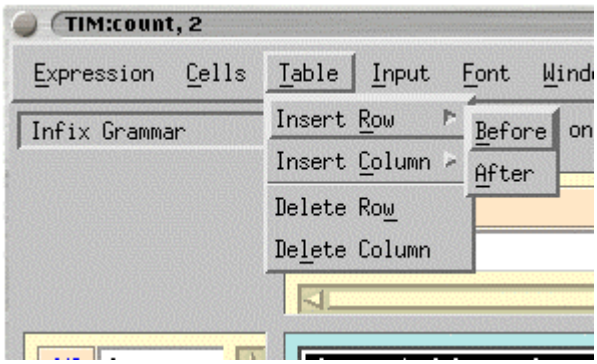


Figure 22: TIM - Insert row before menu item.

6.8 Editing Operations in TIM

Editing operations are supported for entire expressions within the table grid, and the text of the expression.

It is also possible to copy and cut expressions between TIM and other TTS tools – this operation is supported only for the entire expression shown in the window, or for the single selected cell of a table.

TIM uses the TTS protocol function `CSMgrRegisterClip()` to copy expressions. For the purpose of copying a range of cells, it constructs a special clip tabular expression, using the special symbol *TIM Clip F* or *TIM Clip P* to hold a range of cells from either function or predicate table, respectively. This special clip table has a singular grid that matches the dimensions and contents of the copied range.

When the user chooses to copy an entire expression shown (or only a singular cell in a table), its copy (or copy of the contents of the cell) is posted as a clip. In this case no special holding clip tables are used.

As a consequence, the user should not copy a range of cells between TIM and another TTS tool because a range of cells is not truly a proper expression; only an expression fragment and is therefore meaningless. Only proper expressions (entire tables and non-tabular expressions – and also contents of individual cells) should be copied between tools. Note that there is currently no mechanism to forbid the user from actually

copying these special expression fragments to a non-TIM tool, because the clip protocol is global and cannot be restricted to a specific type or class of a tool.

6.9 Allow Using and Defining New Symbols in TIM

The user may have undefined symbols in an expression. Undefined symbols are then shown in the list in the Undefined Symbols Used tab below the input area in TIM. The user can then specify the properties of the symbol, such as its Arity, Tag, FontFamily and Font-Face by clicking in the right column of the symbol entry and changing its value. Once the properties are specified, the user can right-click on the undefined symbol entry and choose the Define Symbol action from the popup window. The symbol is then defined and moved to the All Known Symbols list.

The user can change the properties of symbols already defined by editing the symbol entry under the All Known Symbols tab located below the expression edit area of TIM.

6.10 Tables in Tables in TIM

TIM allows the user to specify a table to be contained within the cell of another table. Conceptually, inner tables can be represented as in Figure 2, where a cell holds another table.

In TIM the inner tables are represented as follows: the cell containing a table shows a special table indicator, composed of the inner table type (symbol) between "<" and ">" symbols. For example, if the inner table is a 2D Normal Function table the cell would contain "<2D Normal F>". To edit the inner table the user double clicks on the cell containing the inner table and a new editor dialog allows the user to edit the inner table (see Figure 23).

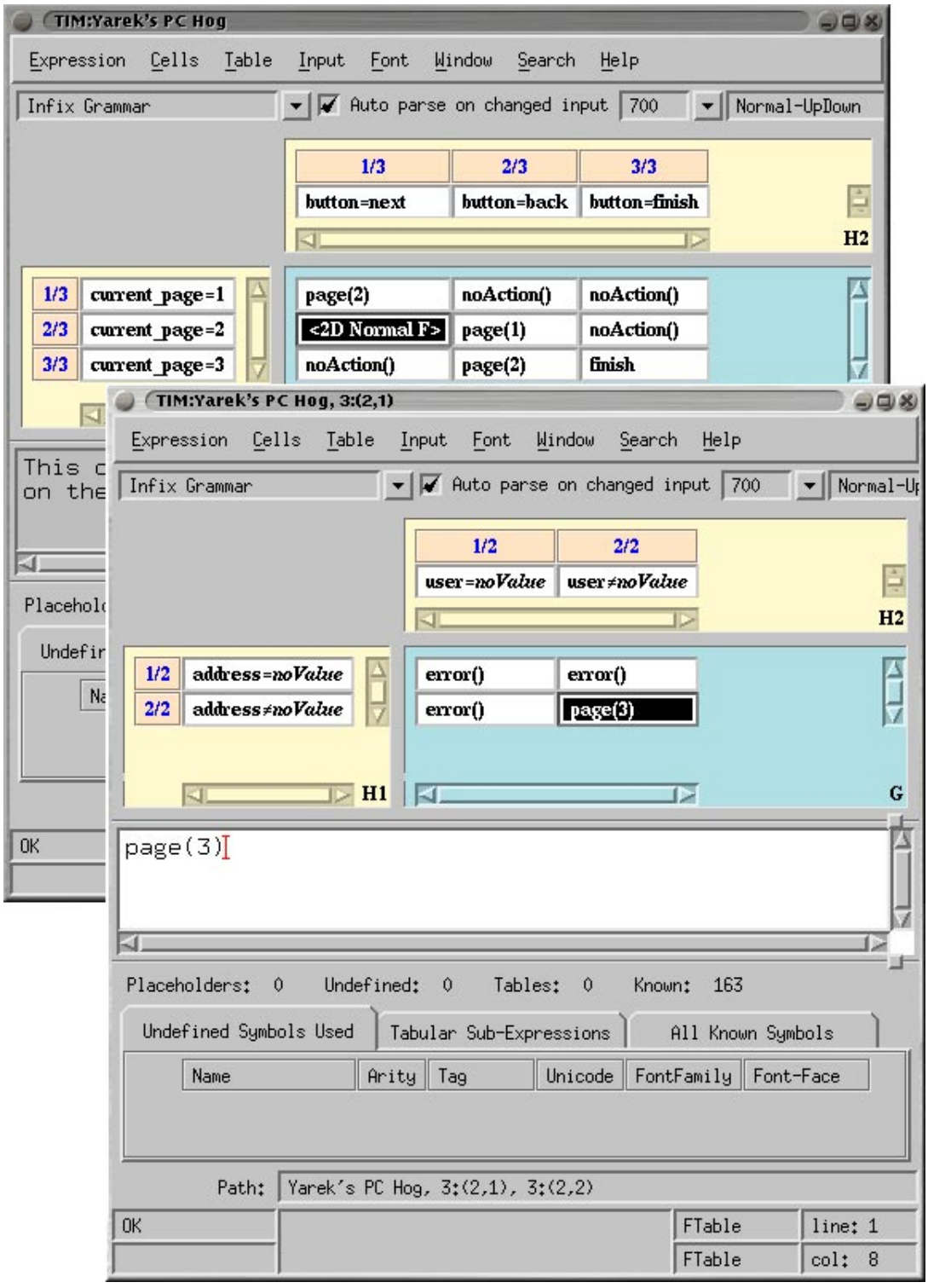


Figure 23: Editing table's inner <2D Normal F> table

6.11 Error Correction in TIM

TIM can be configured to permit the parser to insert tokens that are deemed missing. For example, when the user enters "+" into the empty input field, the parser will determine that the expression is missing left and right terms and the error correction will fill in the missing terms with a placeholder "?" resulting in "?+?". The error correction can also fill in other elements of expressions, such as a missing closing bracket or parenthesis.

The number of corrections performed can be specified in the configuration file, and can be adjusted to each user's preference. The user can specify to have no corrections, but would still be informed about the location of the first error.

The parsers used to process the user input are generated from the grammar definition files described in the Choice of Preferred Notation section above. The grammar file is compiled into an LL(k) parser. An LL(k) parser uses *k* look-ahead tokens to determine the next grammatical rule expansion. At any given point in the parsing of the input string, the parser is aware of what the expected symbols are [23].

If, during parsing, the parser cannot continue because the next token is not in the expected symbol set TIM returns an error. The error indicates where the parsing has failed and what symbols it was expecting but did not find. TIM uses this information to select and inject a "missing" symbol into the input stream, and re-parses it again.

The decision to determine what symbol should be injected is done by an associated grammar error recovery component. A default implementation of the error recovery injects only a missing punctuation symbol. If there is more than one possible symbol to inject, one will be selected according to a pre-determined precedence order. Modifying the error recovery component of a given parser can reprogram and customize this behaviour.

6.12 Extensibility and the Ability to Configure in TIM

TIM can be configured and extended in numerous ways:

1. **Table types.** A new table type can be added to the list of supported types in the TIM configuration file. For each table type the configuration file also specifies the table layout characteristics, such as number and location of grids, and minimum and maximum dimensions of these grids.

2. **Notation and input parsers.** To create a new expression notation the user creates a new notation grammar specification file (see Choice of Preferred Notation section above), compiles it into a Java class, and specifies the parser class name in the TIM configuration file.

3. **Error correction.** When the user compiles grammar specifications into a Java parser, a default error recovery component is created at the time of compilation. The user can customize the behaviour of this component.

4. **Symbol input using keystroke sequences.** Typing symbols into the input area is done through a sequence of keystrokes. The input area actively monitors the keystrokes. When a key sequence is being recognized, the corresponding input text is underlined. When the last key of the sequence is typed in, the underlined portion of the text is replaced by the corresponding symbol (see Figure 24).

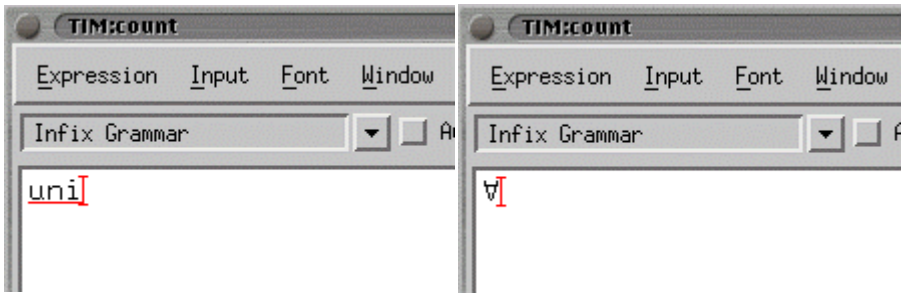


Figure 24: Typing in the ∀ symbol using a keystroke sequence `uni<esc>`

A collection of keystrokes to symbols is kept in a key map file, and the location of key map files can be specified in the TIM configuration file. The user can select the preferred key map from the Input menu.

5. **Valid values for symbol properties.** The user can modify the properties of a symbol, such as Arity, Tag, FontFace, and others. Each property class has a type,

which is *string*, *decimal*, *hexadecimal*, or *enum*. For the *enum* type a list of values is required.

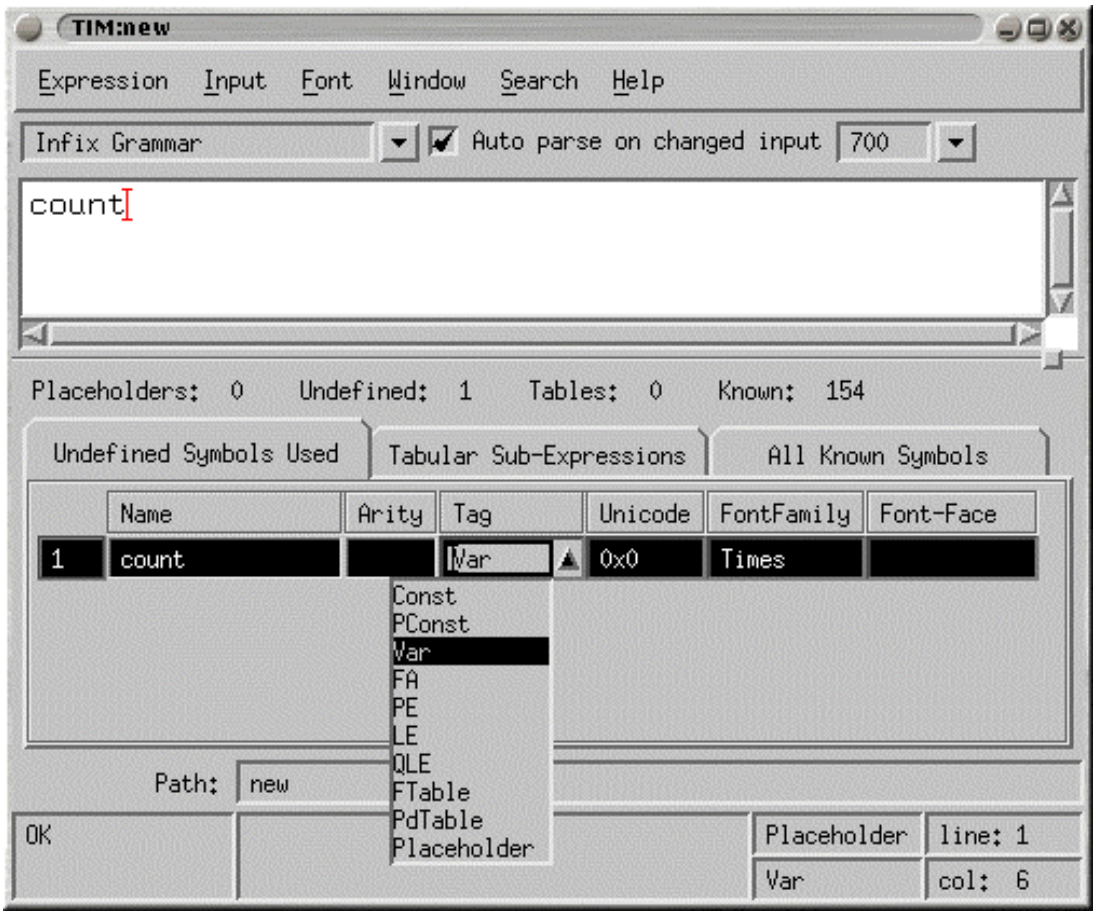


Figure 25: Editing a symbol

When the user edits the properties of a symbol, only valid values are allowed (according to the type). In case of the *enum* type the user can select one of the values listed in the combo-box (see Figure 25).

6.13 Expression Helpers in TIM

TIM employs two techniques to help the user understand the expression evaluation order:

1. For each input notation grammar an HTML document is created showing the grammar in BNF form [22]. With this document the user can deduce the order in which the expression will be parsed.

2. The user can display the expression syntax tree explicitly by invoking the Syntax Tree help tool from the Help menu (see Figure 26).

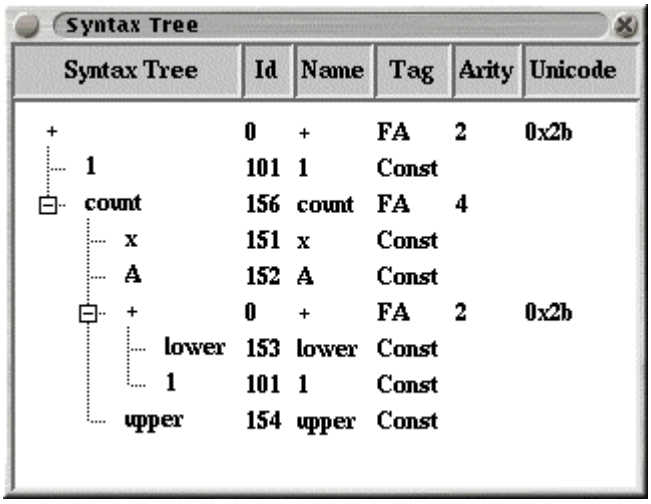


Figure 26: Syntax Tree Dialog

3. For a table, the user can display the Table Legend dialog that shows the table predicate rule, relation predicate rule, and cell connection graph (CCG) class (see Figure 27).

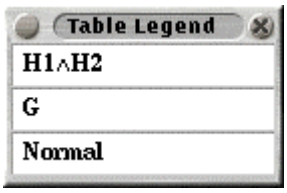


Figure 27: Table Legend Dialog

4. Each grid in a table is clearly labelled, and the cells in the grid are clearly demarcated so that it is clear which cell belongs to which grid (see Figure 23).

5. Rows and columns are also labelled – the row and column headings show the row or column number and the total number of rows an columns in the table. For example, a row label “1/12” indicates that you’re currently showing first row out of total twelve rows (see Figure 23).

7 TTS Environment, Challenges and Solutions

7.1 TTS Environment – An X/Motif based framework

The TTS is an application framework designed for easy integration of components that can present or manipulate tabular expressions. The framework provides basic facilities common to all tools. It consists of three parts:

- Kernel, consisting of essential modules that are used by all other TTS tools. This hides the implementation of the basic TTS data types (TTS objects), which represent expression syntax, semantics and representation.

- Infrastructure - this middle tier contains: tools that operate on individual TTS objects to provide some common service, and modules that allow these tools to be combined. It hides the data structures and algorithms that enable user manipulation of TTS objects.

- Applications – this top tier contains programs that combine the infrastructure modules to manipulate or interpret groups of tabular expressions as documents, which specify or describe some aspect of the computer system. Applications hide the data structures and algorithms that represent and manipulate relational documentation.

The TTS framework is written in C using the X/Motif Graphic User Interface (GUI) components. Any tool or application that integrates this framework must therefore adhere to the protocol and the APIs imposed by TTS and X/Motif.

7.2 Challenges

7.2.1 Expanding GTI is not feasible

An argument can be made that since GTI provides partial desired functionality it might have been easier to expand GTI so that all the criteria are fulfilled. However, expanding GTI would represent a considerable challenge - GTI uses gtk for its user interface and GUI components in gtk have significant limitations:

- Mixing of fonts is not possible – this makes it impossible to represent mathematical formulas that mix Latin and Symbol characters
- gtk also does not have a very efficient way to represent and manipulate large tables
- gtk prevents GTI from being integrated with TTS and take advantage of the expression-creation utilities in the kernel – recreating such utilities would be time consuming and require extensive testing.

TIM, on the other hand, uses ready-made components that are capable of displaying text with mixed Latin and Symbol characters. TIM also uses XRT/table to display very large tables efficiently. All of TIM's components are based on X/Motif that integrate easily with TTS. TIM also takes advantage of the TTS kernel utilities to create tabular expressions.

Therefore, expanding GTI to include capabilities of components used in TIM would have require significantly more work than the work done on TIM itself.

7.2.2 Input String Requirements

In order to concisely express mathematical expressions the expressions must mix Symbol and Latin characters. To display a mixed-symbol string in X/Motif, each section of the string that contains characters from a different set must use a different font matching the set. That is, to display a character from a symbol set a Symbol font must be used.

Another way to mix character sets is to use a string encoding that uses a single font that combines all the different character sets, and a font that can display such an encoding. Such encoding is Unicode, and there are Unicode fonts available for X.

7.2.2.1 Mixing Fonts Solution

When mixing the fonts, one can distinguish symbols not just by the type of character set (Latin or Symbol), but also by the font family (Times, Helvetica, etc.) and font faces (bold, italic, etc). This can have certain advantages:

1. A font family and font face combination could be a distinguishing feature of the symbol. Therefore, there could be two symbols that are identical in every respect, except that one uses the Times Roman font and the other uses Helvetica. TTS, in fact, allows such distinction through the two categories FontFamily and Font-Face.

2. The number of symbols is practically inexhaustible, as long as different fonts with different glyphs exist (or can be invented).

However, there are important usability disadvantages of this method:

1. It requires the user to constantly fidget with the input stream fonts to properly type in the symbol. This can be cumbersome if each consecutive symbol uses a different font.

2. It is error prone – two fonts may not be easily distinguished, for example:

    a. **function(test, test)** – the first "test" is in Arial the second "test" is in Helvetica (sans-serif fonts)

    b. **function(`test`, `test`)** – the first "test" is in Courier the second "test" is in Letter Gothic (serif fonts)

3. Font selection may not be portable across platforms – some fonts that exist on Windows or Mac do not come with, or exactly match, fonts available on Unix/X,

and vice-versa. It is not clear how to deal with a symbol that uses a "missing" font.

7.2.2.2 Unicode Solution

The option of using a single font that contains all the desired characters (Latin and Symbol) is feasible through the use of a Unicode font. The limitations of this choice are:

1. A single font will not be able to take advantage of the font-related symbol categories defined in TTS.

2. Implementation details: Unicode fonts are large and therefore require large computer resources. To rectify the size issue, some Unicode fonts are often made sparse – they do not have all the glyphs.

However, the advantages of using a single Unicode font are:

1. Simplicity – there is no fidgeting with fonts, breaking up input strings into chunks of text delimited by character sets. Uniqueness of the symbol, whether it is a number, letter or a symbol, depends purely on the character id.

2. A very large pool of symbols, far more numerous than those available in the combined Latin and Symbol character sets (when font family and face are ignored).

7.2.2.3 Mixed Unicode Fonts Solution

There is also a third option, and that is to use Unicode-encoded strings with mixed Unicode fonts. This gives uniform character set across all fonts, and it allows us to distinguish symbols by their font family and font faces. The disadvantage is that symbols using similar fonts can be hard to distinguish.

7.2.3 Limitations of the X/Motif Text Widget

A cross-platform portable Motif text widget (XmText) permits the use of single font at a time. Hence, only characters from Symbol or Latin set can be used, but not both.

TCT implementation can, in fact, display mixed-font strings because it employs a DXmText widget – this is a proprietary text widget that uses XmString object for display. XmStrings are Motif's mixed-font strings.

However, a DXmText widget is available exclusively on DEC Unix configured with Motif 2.0, and hence it is not portable to any other platform, or any other version of Motif. All of the current versions of Unix systems ship with Motif 2.1, which has only a single-font XmTest widget. No vendor currently supports Motif 2.0.

One apparent solution was to employ a Unicode font on XmText widget. However, this presented a significant challenge: the display of the text depends on X-based locales that govern the selection of the display font and the input method. Enabling the Unicode locale (utf8) in a portable way across Unix platforms has proved too cumbersome and very limiting, mostly due to lack of input methods availability.

7.2.4 Input Parsing

Input typed by the user must be processed to determine if the input is a valid expression. This requires a parser that can handle the encoding of the input string. Additionally, a framework architecture is required which supports multiple user-defined parsers, each to support a notation.

7.2.5 Ability to Display Tables Efficiently

The set of default Motif widgets do not have a widget that can display data efficiently. The implementation used by TCT uses a RowColumn manager widget holding a set of Button widgets, one for each cell. This is a very inefficient solution – to create a table with more than just a few cells can take a very long time. Adding more rows or columns quickly becomes prohibitive. Unfortunately, this is the only solution that can be achieved using default set of Widgets.

7.3 Solutions to the Challenges

TIM was written using approximately 39,000 lines of commented code using C++, Java and JavaCC. The open-source Yudit component used in TIM has

approximately 58,000 lines of code, of which less than 1% was modified for the needs of TIM.

7.3.1 Yudit – a Portable Locale-free Unicode Editor

In order to overcome the limitations of the Motif text widget and fulfill the requirements for the expression input text that allows mixed Symbol and Latin characters, a solution was found: Yudit [24].

Yudit is a free-license portable X/Motif based application that allows the user to create documents in Unicode encoding written in C++. It uses its own special widget to handle the display of the text, and it does not require locale-based input methods to allow the user to create a document – it includes its own input method that can be customized for each user. The only limitation is that it can use only one Unicode font at a time.

7.3.2 Translating Unicode to Motif Strings

Without the proper locale installed on the Unix system the Motif widgets are unable to display Unicode encoded text. Since TTS is a Motif application and uses Motif widgets to display the text, TIM provides a facility to translate the expression between its Unicode and XmString representation.

7.3.3 XRT/table – a Widget for Displaying Tables in X/Motif

To overcome the limitations of the default Motif widget set, TIM employs add-on commercial Motif widgets, XRT PDS [25], produced by Sitraka Inc. To display a tabular expression TIM employs multiple instances of the XRT/table widget – one for each grid in the table. TIM combines these separate grid-holding widgets in such a way that scrolling, row and column sizing are synchronized across the different grids – the rows and columns in each grid are always properly aligned.

TIM also uses other XRT components to build a more useable GUI. Most notably, the outliner widget is used to display the syntax tree in the Syntax Tree Dialog. The tab manager component is used to organize a list of different classes of symbols.

All XRT widgets display strings in the XmString format, and therefore rely on the TIM's Unicode to XmString expression translator to display the expression text.

7.3.4 Java TTS Expression Parsers

By employing Unicode, TIM has simplified input text manipulation. The remaining challenge was to parse the input string.

Unicode encoding (utf8) lends itself for easy manipulation in Java. Therefore, TIM uses Java-based parsers to parse the input and produce a syntax tree in Java. TIM then traverses the Java syntax tree on the native side through the JNI interface. A corresponding TTS-expression object is created.

The Java parsers for the input are specified in the syntax description files and translated into JavaCC [26] parser specification files. JavaCC compiles the parser specification into a set of Java source files which, when complied into class files, produce a working parser.

Java loads its classes at run time. This fact has greatly simplified the construction of the parser framework that allows the user to create custom-made input parsers. To add a new parser, all the user has to do is create an input parser specification file, compile it, indicate in the TIM configuration file the name of parser's main class, and restart TTS. All the parsers indicated in the configuration file, including the user-specified parser, will be loaded the next time TTS is started.

8 Future work

TIM meets all the effective tabular expression editor criteria. There is, however, room for improvement.

8.1 Evaluation by Users

The users should validate TIM's effectiveness. Only time and usage will tell whether there are any more criteria for effective tabular editors, and whether any rebalancing of the implementation of the features is needed. A thorough study should help discover the limitations of the TIM's approach to constructing tabular expressions and suggest further improvements.

8.2 Finer Control Over Error Correction

First, there is a need for finer control over error correction. Automatic error correction is guesswork, and the corrections may not be what the user intended. Currently, when the correction is done the user may not clearly understand what has changed in the input – changes are not indicated in any way.

To improve this, the tool should warn the user or highlight changes made, and allow the user to undo part or all error-corrections made. This will give the user an opportunity to double-check that the automatic corrections agree with her intent.

8.3 Keyboard Shortcuts to Activate Actions

Currently there are no keyboard shortcuts defined to access menu actions. This issue needs to be resolved, but the solution must take into the account the possibility that the user-defined keyboard mappings for generating symbols in the input may interfere with some of the shortcuts. This can be resolved either by ignoring mappings for

reserved keywords, or ensuring that the shortcuts use a special prefix unlikely to be used in the keyboard mapping (for instance, use of Alt key).

8.4 Improve Expression State Indicators

Expand expression state indicators to show pairs of parentheses or brackets – employ colours, subscripts or different font faces. This would be helpful to visually navigate the input, and delimit sub-expressions.

8.5 Migrate Utilities into the TTS Framework

The input parser framework and the parser-based syntax tree to input text translator is a useful tool that can be used elsewhere in TTS. TIM's symbol editor could also be made into a separate component that can be used elsewhere in TTS. These elements of TIM should be migrated into the TTS framework so that they are accessible by other tools as part of the TTS services layer.

8.6 Modularization and Code Cleanup

Re-modularize the application. TIM is a research project and many components were added at a time when their impact may not have been fully understood. Many ideas and user interactions where not fully understood until the user interface has been created. In the process, module boundaries have become somewhat blurred and their interfaces have become less clear-cut.

Now that the purpose of all the elements is better understood, the implementation could be re-factored better to clarify the module boundaries. Also, some of the design decisions should be revisited and re-implemented. In particular, the editor should be more effective at using inheritance and polymorphism to better implement the tabular and non-tabular editing modes.

9 Conclusions

To create tabular expressions quickly an effective tabular editor is needed. Although it is possible to create tabular expressions using a basic text editor for the expression input and a parser for processing the input, many users will find this tedious and error-prone. The average user requires an environment that will make the task of creating tabular expressions more easily, and with more immediate feedback.

9.1 Effective Tabular-Expression Editors – Design Principle Tradeoffs

In order to be effective, tabular-expression editors must meet the outlined criteria. However, all the criteria work in unison and therefore affect each other. By necessity, there are some trade-offs, limits and dependencies for each criterion:

1. **Correct expression** – the tool must create correct tabular expressions. In TIM this depends greatly on the validation of the input, which in turn depends on the selected parser. If the parser is not correct the validation may be faulty, causing an incorrect expression to be created.
2. **Verification and error analysis** – in TIM if an incorrect expression is loaded, an error will be printed, a log trace generated, and the expression rejected outright. TIM does not attempt to proceed despite the error.
3. **Free input with flexible validation** – in TIM the user can type in the expression freely using a keyboard; syntactic validation can be done either while the user is typing or when explicitly specified by the user. This depends on the correct implementation of the parser – if the parser is incorrect, the input will be validated incorrectly.
4. **Choice of preferred notation** – in TIM this depends on each notation parser being implemented correctly.

5. **Expression state indicators** – in TIM the expressions use a placeholder to indicate missing sub-expressions. To enable automatic display of the placeholder symbol "?" the error recovery option must be used. By default, the error recovery will also attempt to insert "missing" right closing parentheses, a comma separator and a right closing bracket. This behaviour can be modified by programmatically modifying the Java error recovery class – this requires modification and recompilation of one Java source file, per each parser. For the change to take effect TIM needs to be restarted (to reload the newly changed Java class file).

6. **Configurable table types** – future new table types can be easily added without requiring any source code changes. Current implementation assumes that the grids of a table are themselves arranged in a super-grid, and that the dimensions of grids coincide. That is two grids located one above the other will have the same number of columns, and two grids side by side will have the same number of rows.

7. **Insertion and deletion of cells** – TIM allows for the insertion and deletion of rows and columns of cells into a table, but only where it is permissible. In particular, some grids may have restricted dimension, and cannot grow or shrink beyond a specific size. Also, insertion of rows or columns will happen across the neighbouring grids.

8. **Editing operations** – common editing operations, such as cut, copy and paste, are implemented in TIM. Expressions can be pasted into cells, and cells can be copied. The limitation of copying cells is that the source and target ranges match: it is impossible to copy a single cell into a range of cells at one go – each target cell must be pasted individually.

9. **Allow using and defining new symbols** – in TIM symbols that are not yet defined can be used. This is only allowed within the single cell. That is, TIM will not permit the user to switch to another cell, unless all the symbols within the current cell are defined.

10. **Tables in tables** – tables are valid expressions; TIM allows the user to create tables in the cells of tables as long as the cell is undefined. If the cell is defined,

the user must first clear it before creating a table in the cell. A design decision was made not to visually show tables in a cell, but instead to display a separate window for the inner table – displaying inner and outer tables in a single table has the potential of creating very large windows, which makes them awkward to handle. However, separate widows for each inner table can result in lots of windows if there are many inner tables. A more elegant solution to this presentation problem is required.

11. **Error correction** – in TIM simple syntax errors are corrected automatically. The behaviour can be turned off completely. The user also has an option to modify the error correction class (written in Java) to restrict or modify which corrections take place. To take effect, the entire TTS must be restarted (but not recompiled).
12. **Extensibility and the ability to configure** – the user can modify the behaviour of TIM. Care must be taken as a single change can have profound effect on how expressions are constructed. Care must be taken, particularly when sharing expressions among different users.
13. **Expression helpers** – TIM displays the syntax tree only for the current cell. The syntax tree will only be updated when the expression is validated. If the parsing is turned off, the syntax tree display will show the expression at a different state than what the current input string is apparently showing (since the input is yet to be validated).

TIM is effective because the user can create tabular expression for use in TTS quicker than TCT or any other surveyed tool. The level of effectiveness of TIM can be further ascertained by conducting a thorough user-interaction study. Such study would use separate groups of users whose tasks would be to construct certain tabular expressions using the described tools.

9.2 Fulfilling All Criteria is Feasible, Balancing Is Needed.

We have demonstrated that none of the existing word editors, spreadsheet editors, equation editors and two custom table editors meet the criteria. None of the surveyed applications are effective tabular expression editors.

We have also demonstrated that it is possible to create an effective editor that does meet all the criteria. However, the criteria have interdependencies. To make the tabular expression truly effective the fulfillment of all the criteria must be done with care to properly balance off the effects of the individual features.

9.3 Limitations inherent to TTS

The work on this thesis has allowed for a thorough examination of the TTS design. There is room for improvement:

- The API for manipulating Expressions is cumbersome – manipulation of expressions requires the usage of too many Path and Expn objects, which can become difficult to track.

- Lack of infrastructure to protect object ownership – it is difficult to track which tool is modifying which expression, expression part, or symbol. It would be beneficial to have object read/write locking that would prevent tools from interfering with each other when working on an expression.

- Rethink the concept of project (context) management and presentation. Current implementation is non-intuitive, and a challenge to the novice user – to access any tool the user must first open the context (project would be a better name). The most confusing part is that some tools can only be invoked when the appropriate selection is made, but not very clear what that selection is.

- Consider using Java for next implementation of TTS – cheaper UI components, wider platform support, and the advantage of using object-oriented technology. The current reliance of X/Motif limits its use to the Unix environment and prevents TTS from being used on Macintosh and Windows.

  As far as tabular expressions go, there are a few improvements that can be made:

- Introduce expression inheritance. It would allow creation of expressions that are like another expression, except for a few changes, such as addition of a few extra rows and a change in one of the existing cells.

- Add ability to access expressions by reference, not just copy. Currently it is impossible to refer to an existing expression elsewhere without making a copy of the expression. Expressions should have a reference name. This reference name should be used just like any expression, and the user should be able to follow the references to see the reference target expression.

9.4 TTS is not an implementation medium

The purpose of creating TIM was to explore the concepts that make tabular expression editors more effective. There is a certain level of experimentation and subjective judgment involved in creating an effective user interface. This experimental process requires frequent changes and modifications to the user interface code.

Currently, the tabular expressions cannot be translated directly into working code. Therefore, tabular expressions are not currently an effective means for implementing experimental user interface prototypes. However, once the prototype user interface look-and-feel has been agreed on, tabular expressions can and should be used to design a more complete solution.

It should also be noted that before TIM there was no expression editor that would allow the user to quickly create and edit expressions. The process of creating the design using tabular expressions was therefore very tedious, cumbersome, ineffective, and even infeasible as some tabular expressions cannot be constructed using the existing tools.

9.5 Tabular Notation – It Would Have Been Beneficial

Table notation as a means of documenting software could have been used in the process of designing the TIM tool. Many of the design decisions ware made during the development process and, although the ideas are sound, the implementation is not clean and sufficiently robust. Much of the unclean design could have been avoided if tabular notation was used at the design phase, from the start of the project. It would have helped to clarify some issues that remained undiscovered until the implementation was already in progress.

Furthermore, as the implementation progressed, the software documentation that went along with it could have been maintained in tabular form. This would give an opportunity to the designer, at the end of the project, to review the implementation and decide if it matches the intended design. The final document produced at the end of the implementation would have served as a reference manual for future developers who needed to upgrade, change or simply re-implement the tool.

Unfortunately, the limited functionality of the existing tools for creating software documentation using tabular notation would have prevented the author from creating effective TIM documentation in a reasonable amount of time. As the effectiveness of tabular-expression editors increases, so will the use of tabular expressions as a means for precise software documentation. It is the hope of the author that the contribution of this thesis will bring us closer to this goal.

References

[1] D.L. Parnas, G.J.K. Asmis, and J. Madey, “Assesment of Safety Critical software in nuclear power plants”, *Nuclear Safety, vol. 32, no. 2, pp 189-198, April-June 1991*

[2] N.G. Leveson, C.S. Turner, “An investigation into of the Therac-25 accidents”, *IEEE Computer, 26(7), pp.18-41, July 1993*

[3] K.L. Heninger, J. Kallander, D.L. Parnas, J.E. Shore, “Software Requirements for the A-7E Aircraft”, *NRL Memorandum Report 3876, United States Naval Research Laboratory, Washington, DC, 523 pgs, Nov. 1978*

[4] W. W. Gibbs, “Software's Chronic Crisis”, *Scientific American, September 1994, pp. 86-95*

[5] I. Peterson, “Fatal Defect: Chasing Killer Computer Bugs”, *Random House, Toronto, 1995*

[6] D.L. Parnas, “Software Aging”, *ICSE16 Proceedings*

[7] S.D. Hester, D.L. Parnas, D.F. Utter, “Using Documentation as a Software Design Medium”, *Bell Systems Technical Journal, Vol. 60, No. 8, October 1981, pp. 1841-1977*

[8] D.L. Parnas and J. Madey, “Functional documentation for computer systems engineering (version 2)”, *CLR Report 237, McMaster University, Hamilton, Canada, Telecommunications Research Institute of Ontario (TRIO), Sept 1991*

[9] D.L. Parnas, “Inspection of Safety-Critical Software Using Program-Function Tables”, *IFIP ’94 Proceedings*

[10] D.L. Parnas, “Tabular representation of relations”, *CLR Report 260, McMaster University, Hamilton, Canada, Telecommunications Research Institute of Ontario (TRIO), October 1992*

[11] R. Janicki, “Towards a Formal Sematics of Parnas Tables”, *17th International Conference on Software Engineering, IEEE Computer Society,* Seattle, *WA, April 1995 pp. 231-240*

[12] D. Peters, “Generating a Test Oracle from Program Documentation”, *CLR Report No. 302, Telecommunications Research Institute of Ontario (TRIO), April 1995*

[13] McMaster University Software Research Group, “Table Tool System Developer’s Guide”, *CLR Report No. 339, Telecommunications Research Institute of Ontario (TRIO), pp. 86, January 1997*

[14] L. Zhang, “A Template/Overlay Approach to Displaying and Printing Tables”, *M.Eng. Thesis, CLR Report No. 289, Telecommunications Research Institute of Ontario (TRIO), pp. 122, June 1994*

[15] W. Li, “Table Construction Tool”, *CLR Report No. 330, Telecommunications Research Institute of Ontario (TRIO), pp. 99, July 1996*

[16] H. Shen, “Implementation of Table Inversion Algorithms”, *M.Eng. Thesis, Department of Electrical and Computer Engineering, McMaster University, Hamilton, Canada, December 1995. Also printed as CLR Report No. 315, Telecommunications Research Institute of Ontario (TRIO), December 1995*

[17] Microsoft Word 2000, *Microsoft Inc., 1999*

[18] FrameMaker 5.5, *Adobe Systems Inc., 1997*

[19] Interleaf 6.1.1, *Interleaf Inc., 1995*

[20] Microsoft Excel 2000, *Microsoft Inc., 1999*

[21] I. Vulanovic, M. von Mohrenschildt, “A Grand Table Interface – Specification / Developer’s / User’s Guide”, *CLR Report No. 352, Telecommunications Research Institute of Ontario (TRIO), August 1997*

[22] C.N. Fischer, R.J. LeBlanc Jr., “Crafting a Compiler with C”, *The Benjamin / Cummings Publishing Company, Inc., pp. 91, 1991*

[23] C.N. Fischer, R.J. LeBlanc Jr., “Crafting a Compiler with C”, *The Benjamin / Cummings Publishing Company, Inc., pp. 134, 1991*

[24] Yudit 1.5, *Gaspar Sinai (GNU General Public License), 1999*

[25] XRT PDS 2.0, *Sitraka Inc. (formerly KL Group Inc.), 2000*

[26] JavaCC 1.1, *Sun Microsystems, 1999*

Appendix A: Table Input Method User's Manual

Table Input Method (TIM) is a TTS tool that allows the user to construct tabular expressions. This appendix describes how to use and configure TIM.

A.1 Typographical Conventions Used in this Manual

To make the contents easier to understand we will use the following typographical notations:

`Typewriter Font`	• Letters and symbols that can be typed directly from the keyboard • Source code and contents of configuration files • Commands that you enter at the terminal window • Names of modes, such as the name of the input keymap
Italic Text	• Pathnames, filenames, program procedure and parameter names • New terms as they are introduced, and to emphasize important words
Bold Text	• Names of GUI components such us buttons and menus
<Esc> <Ctrl-a>	• Keyboard keys and key combinations

A.2 Unicode Fonts Are Required

The X font server for your display must have the necessary Unicode fonts for TIM to function properly. Copy the Unicode fonts in *$TTS_HOME/tim/font/unicode* directory so a location accessible by your X font server, say */usr/lib/X11/fonts/unicode*. Then, either add */usr/lib/X11/fonts/unicode* to your X server font path, or configure the font path manually using this command:

```
xset +fp /usr/lib/X11/fonts/unicode
```

To check if you have the right fonts, check if you can display the Unicode font with the following command:

```
xfd -font "-misc-fixed-medium-r-normal--20-*-*-*-*-*-iso10646-1"
```

Once you successfully configure your X server to find the Unicode fonts you are ready to proceed.

A.3 Java 2 is required

TIM requires Java 2 to properly run TIM's input parsers.

A.4 Getting Started

TTS allows you to create to manipulate software documentation in form of tabular expressions. A TTS document is composed of a number of tabular expressions.

To open a TTS document first you must open a new Context window by selecting **Tools→Context Manager** from the Table Tool System menu. Then, from the Context window menu, choose **File→Open…** menu and select the TTS document you wish to work on. TTS document files have *.tts* extension. You can create a new TTS document in the current Context window by selecting **File→New** menu in the Context window.

In the Context window you can create a new expression by selecting **Expression→New Expression** menu. You will then be presented with the Expression Name dialog – type in the name of the expression and click on the **OK** button. The newly created expression is now listed in the Context window. You can rename the expression by selecting **Expression→Rename** from the menu.

To launch a TTS tool for an expression, select the expression in the Context window, click the right mouse button and choose the desired tool from the popup menu to launch it (Figure 28). To edit the expression in TIM, choose **Table Input Method**.

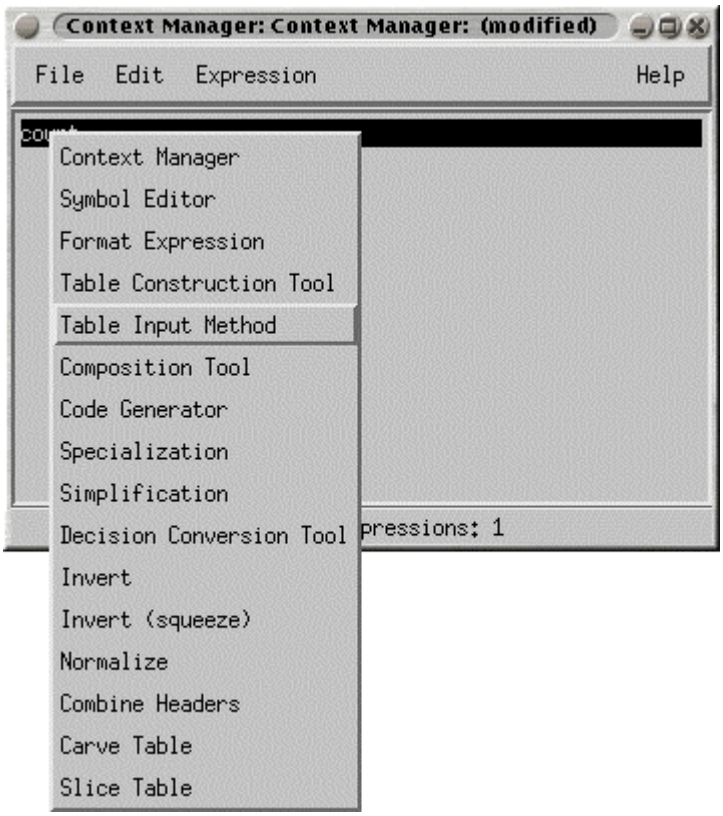


Figure 28: Selecting a tool in TTS

A.4.1 Editing a New Expression

If you launch TIM for newly created application, you will need to select the type of expression you want to create. You will be presented with Type Selection dialog (Figure 29). The expression type you can choose can be either *tabular* or *non-tabular*.

A choice of non-tabular expression means that the *root* symbol is *not* a table. You will be permitted to create tables, but only as sub-expressions of the non-tabular root.

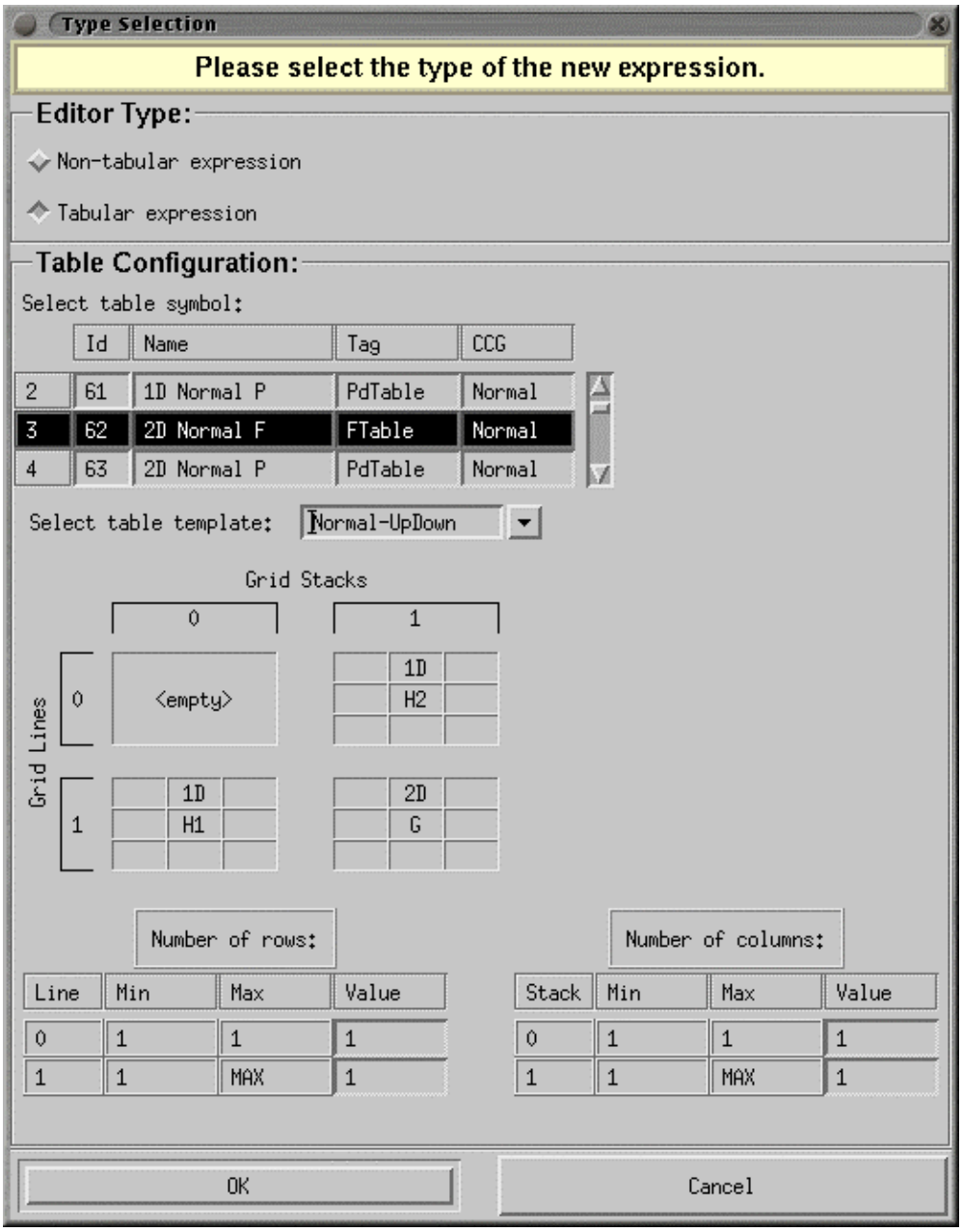


Figure 29: New expression type selection dialog

If you choose to create a tabular expression, you need to make the following additional configuration choices from the dialog:

- Root *table symbol* by selecting appropriate symbol from the table symbol list
- Table *template*, which governs the table grid location and orientation, from the template combo box.

- *Dimensionality* of the table grids by clicking on the grids in table sample area and toggling their state. The number of available grids will change depending on the type of symbol table and table template selected. Some grids can be toggled so that they don't appear at all, while others cannot be altered altogether.
- Number of rows and columns in each line[1] and stack[2] of grids. This ensures that two neighbouring grids agree on the number of rows (or columns) where they meet. Some stack line values are restricted and cannot be changed. None of the values can be less than 1.

After making the appropriate choices click on the **OK** button to start editing expression in TIM.

A.4.2 Expression Symbols

Symbols are the basic building blocks of expressions. An expression is a collection of symbols organized in a tree-like structure. Each node in a tree is a symbol, and the relationship between a node and its child-node is determined by the properties of the symbols.

Property	Type	Value	Description
Name	Presentation	String	Name of the symbol. Symbol name can be used for presentation only if Unicode is not set.
FontFamily	Presentation	String	Name of the font family used to render the symbol, such as *Times, Courier, Symbol,* and more.
Font-Face	Presentation	Decimal Integer	The face of the font to use to render the symbol, such as *Normal, Bold, Italic,* and more (font faces are encoded as integers).
Tag	Structure / Presentation	Decimal Integer	The type of symbol, such as *Constant, Variable, Function Table,* and more (tags are encoded as integers). Some tags also govern the presentation, for example Function Table indicates to use a tabular display.
Arity	Structure / Presentation	Decimal Integer	This property determines how many children the symbol must have in an expression.
Unicode	Presentation	Hexadecimal Integer	This property, when set, indicates the index used to retrieve Unicode glyph.

Table 11: Basic symbol property classes

[1] We reserve term "row" to describe only a row of cells, and use term "lines" to describe rows of grids.
[2] We reserve term "column" to describe only a column of cells, and use term "stack" to describe columns of grids.

There are currently 23 separate types of properties, also known as *symbol information classes*, such as *Name*, *FontFamily*, *Arity* and more. Some of the properties are used for presentation and display only, some are used to determine how to assemble symbols into expressions, while others hold information needed to construct test oracles in C.

Typically, a symbol defines only a subset of properties. The property most needed for the purpose of presentation and organization is listed in Table 11. A presentation property, such as FontFamily, is used to display or render the symbol. An organization property, such as Arity, is used to determine how the symbol will be used in relation to other symbols in an expression.

TIM uses all the symbol properties listed in Table 11, though some components of TIM may use only some of the properties to render the symbol.

A.4.3 Rendering expressions

TIM components render expressions using either Unicode strings or XmStrings (Motif multi-font string). To create a string representation of an expression TIM walks the expression syntax tree (aided by the currently selected input parser) and constructs the string piece-by-piece from individual symbol strings.

A symbol string is constructed using the symbol's Unicode property. The Unicode property value can be converted to either a Unicode string or an XmString. If the Unicode property is not set, the Name property is used instead. The Name property always uses Latin-1 characters, which is compatible with either Unicode string or XmString.

To convert the Unicode property value to an XmString, TIM uses a special Unicode-to- Motif mapping. That is, a Unicode character maps either to a character in Latin-1 font or a character in Symbol font. To complete the Unicode to XmString conversion TIM uses the symbol's FontFamily and Font-Face properties to select an appropriate X fonts of the XmString.

A complete set of available mappings can be found in Appendix G: Useful Unicode Symbols below. To understand how the parser aids in the creation of the expression string see Appendix C: Translating Stored Expressions to String below.

A.5 TIM Anatomy

TIM provides the user with the interface to build tabular expressions. The window is composed of the menu bar, toolbar, table area, input area, symbol information area, and status bar (Figure 30).

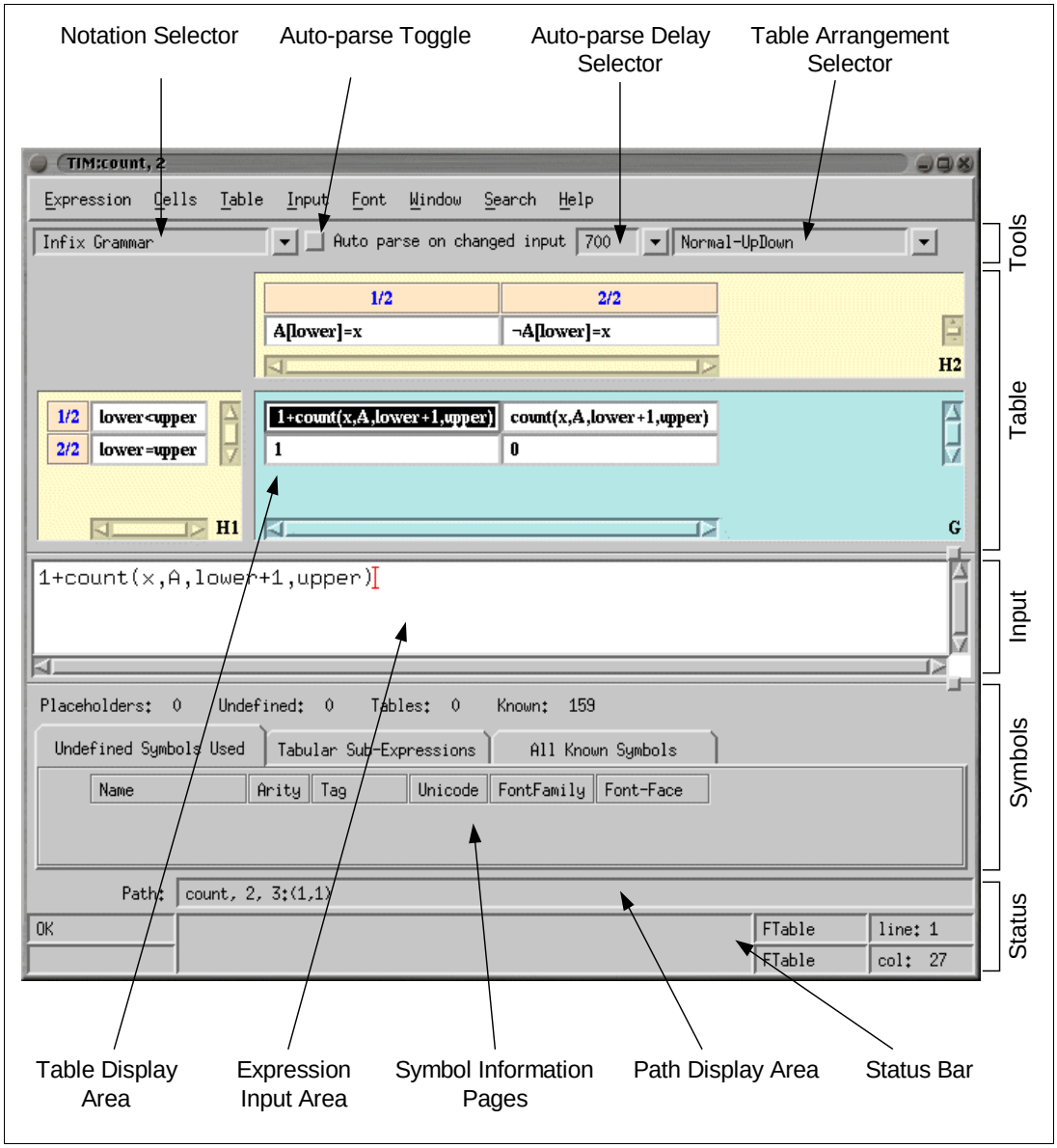


Figure 30: Components of TIM

A.5.1 Menu

- **Expression menu.** Contains actions affecting the entire expression:

Revert changes	Revert all expression changes since the last time changes were applied.
Apply changes	Save all changes made to the current expression.
Cut	Make a copy of the entire expression and clear the expression. The current expression is cleared and new one is created - you will be prompted to specify the type of the new expression.
Copy	Make a copy of the entire expression
Paste	Replace the entire expression with a copy of an expression held in the expression clipboard.
Clear	Clear the expression (and create a new one).
Close	Close the window. You will be prompted to apply any outstanding changes made to the current expression.

- **Cells menu.** Contains actions for manipulating ranges of cells. Actions are:

Cut	Copy the contents of the selected range of cells and clear the cells
Copy	Copy the contents of the selected range of cells
Paste	Paste the previously copied range of cells
Clear	Clear the contents of selected range of cells
Revert changes	Revert all changes made to the selected range of cells since the last time changes were applied.

- **Table menu.** Contains actions to insert or delete rows and columns in a table:

Insert Row:	
Before	Insert a row before (above) the current row.
After	Insert a row after (below) the current row.
Insert Column:	
Before	Insert a column before (to left) the current column.
After	Insert a column after (to right) the current column.
Delete Row	Delete the current row.
Delete Column	Delete the current column.

- **Input menu.** Select preferred input keymap. By default the menu contains:

TTS-input	Default input keymap
TTS-Unicode	Alternate input keymap

The contents and the selection of keymaps can be changed, see sections A.6 and A.7.1 for more details.

- **Font menu.** Adjust the properties of the current input area font:

Size	Font size
Weight	Font weight (medium, bold, …)
Slant	Font slant (roman, italic, …)
Spacing	Spacing (mono-space, proportional, …)
Avg. Width	Width in dots-per-inch (50, 60, 110, …)

- **Window Menu.** Adjust window properties:

Tab Size	Change the input area tab size.
Editor Colours	Change foreground and background colours of the input area.
Main Grid Colours	Change the shading and label colours of the main grid area.
Header Grid Colours	Change the shading and label colours of the header grid area.
Label Colours	Change the foreground and background colours of the row and column headers
Cursor Colour	Change the colour of the cursor in the input area.

- **Search menu.** Actions that operate on the contents of the input area:

Search/Replace	Display a dialog to type in a search string, and optionally replace the matched string with a new one.
Go To	Go to the specified line and column in the input area.

- **Help menu.**

Show Syntax Tree	Displays a dialog that shows the syntax tree of the expression currently being edited in the input area.
Show Table Legend	Displays the table legend containing the table's CCG, predicate and relation rules.
About	Show the information dialog about TIM.
Help	General application help.

A.5.2 Toolbar

Toolbar contains the following tools:

Notation Selector	Click on the arrow button to the right of the filed to display a list of supported notations, and select the preferred notation.
Auto-parse Toggle	When the checkbox is on, parsing of the input area will be engaged once the contents have changed *and* the auto parse delay has expired.
Auto-parse Delay Selector	Specify how long to delay (in milliseconds) before engaging auto-parse after input change. Click on the arrow button to the right of the filed to display a list of default delay values and select one entry.
Table Arrangement Selector	The table type may have alternate grid arrangement. Click on the arrow button beside the field and select preferred grid arrangement.

A.5.3 Table Area

The table area displays the tabular expression. If you're not editing a table, it will not be shown. Each table is composed of a number of grids and each grid contains a set of cells.

There are two types of grids in each table: main grid and headers. The main grid is labelled *G*, and the header grids are labelled *H1*,…, *Hk*. The grid labels are shown in the bottom-left corner of each grid. The areas of header grids are shaded in a different colour from that of the main grid.

A grid may contain more cells than can be currently displayed. You can scroll the grid to display any hidden rows and columns by using the scrollbars. Grid scrollbars are synchronized, so that the rows and columns in the neighbouring grids remain properly aligned. For example, when you scroll the main grid horizontally to show column 10, the grid above (and below, if such grid exists) will also scroll to show its column 10 directly above (and below) column 10 in the main grid. Scrolling of rows is analogous

Rows and columns are labelled. The label of a row (or column) is shown as *k/n*, where *k* indicates the row (column) number and *n* indicates total number of rows

(columns). For example, label *3/5* indicates that you're looking at third row out of 5 rows total.

Rows and columns resize automatically to the largest cell. This is to ensure that cells always show complete contents.

To edit a cell, click on it to select it. The current cell is shown in reverse colour and is always selected. To select a range of cells click on the first cell and drag the mouse over the range of cells. A range of selected cells is also shown in reverse colours. To distinguish the current cell from other selected cells, the current cell has an outline in the colour of the text.

The colours of the input area, cell contents, grid areas, and row and column labels can all be changed through the relevant actions in the Window menu.

A.5.3.1 Inserting New Rows and Columns

You can insert rows of cells provided you have not reached the row limit for a given line of grids by selecting **Table→Insert Rows→Before** or **Table→Insert Rows→After** menu.

The rows of cells can be inserted either before (above) or after (below) the current row. If you have selected more than one row of cells you will insert a corresponding number or rows. That is, if you have selected a 2-rows by 4-columns range of cells, and you select you will insert two rows before your current selection. In the similar fashion you can insert columns of cells.

Note that some grids cannot be altered. For example, the header grid H1 of the Normal 2D F table must always have 1 column of cells. Grids can be configured to require a minimum (greater or equal to 0) and a maximum (up to MAX_INT) number of rows and columns. The neighbouring grids in a line (stack) of grids are guaranteed to have identical number of rows (columns).

A.5.3.2 Operations for Ranges of Cells

When you select a range of cells, you can perform the following actions on that range:

Cut	Copy the contents of the selected range of cells and clear the cells
Copy	Copy the contents of the selected range of cells
Paste	Paste the previously copied range of cells
Clear	Clear the contents of selected range of cells
Revert changes	Revert all changes made to the selected range of cells since the last time changes were applied.

If you have selected a *single* cell, the copy cell can be pasted to replace an entire expression in a TIM window.

For example, suppose you wanted to create a new expression that is a copy of the cell G(1,1), you would then perform the following steps:

1. Select cell (1,1) in grid G
2. Select **Cells→Copy** menu
3. Create a new expression in Context window, open the new expression (choose any type of expression in the Type Selection dialog)
4. Paste the expression using **Expression→Paste** menu

If you copy a *range* of cells you can only paste them as a *range*.

A.5.3.3 Displaying the Table Type Legend

To clarify what type of table you're currently editing you can display the Table Legend (see Figure 31) by selecting **Help→Show Table Legend** menu.

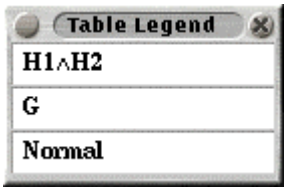


Figure 31: Table Legend dialog for a Normal 2D table

The legend is composed or three fields that display in order:

1. Predicate rule
2. Relation rule
3. CCG case

The rules notation depends on the preferred input notation currently selected.

A.5.4 Input Area

To create an expression you type the expression text in the input area. You can type any Unicode character symbol (including regular ASCII) in the input area – see section *A.6 Typing expression text* below for details.

A.5.4.1 Input Validation

The input area validation depends on two factors:

1. Choice of notation. Use Notation Selector to choose your preferred notation.
2. Whether auto-parsing is engaged. Check on the Auto-parse Toggle and select parsing delay using Auto-parse Delay Selector.

Your input will be validated according to the notation you have selected. The input parsing is engaged when auto-parsing option is turned on or when you move to the next cell. When you select the auto-parsing option, the input will be re-parsed every time you change it, but only after the specified time delay has expired.

If during the input parsing the parser determines that at some point the input is missing a sub-expression, a placeholder symbol "?" will be injected at that point and the parsing will continue. The parser will also automatically inject missing closing brackets and parenthesis. This symbol injection behaviour can be tuned – see section B.2 Custom Error Correction below.

When an error is found in the input the details of the error will be shown in the error dialog window (Figure 32). The status bar at the bottom of the window will also be updated to show the *Error* status, and the error message.

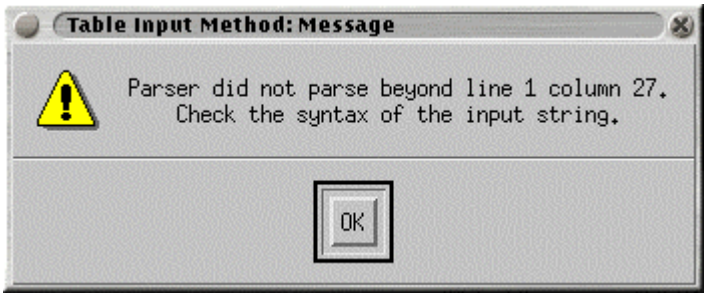


Figure 32: Parsing error message dialog

When the parser finds a new, previously undefined symbol in the expression, it will list it in the *Undefined Symbols Used* tab in the Symbol Information Area. To learn how to define symbols see section *A.5.5.1 Defining new symbols* below.

A.5.5 Symbol Information Area

The symbol information area is divided into three tabs. Each contains information about specific class expression symbols:

- Undefined Symbols Used
    - Show new undefined symbols
    - Allow the user to edit the properties of the new symbol and define it
- Tabular Sub-Expressions
    - Display a list of table symbols that represent roots of tabular sub expressions. If expression changes, but still uses the same table symbol, it will retain the associated table sub-expression contents.
    - Right-click on the table symbol listed to edit the table in a new TIM window.
- All Known Symbols
    - Allow the user to edit the properties of known symbols.

A.5.5.1 Defining new symbols

When you type a new expression it may be using expression symbols that are not yet defined in TTS. All undefined symbols are listed under the *Undefined Symbols Used* tab in the *Symbol Information Area*.

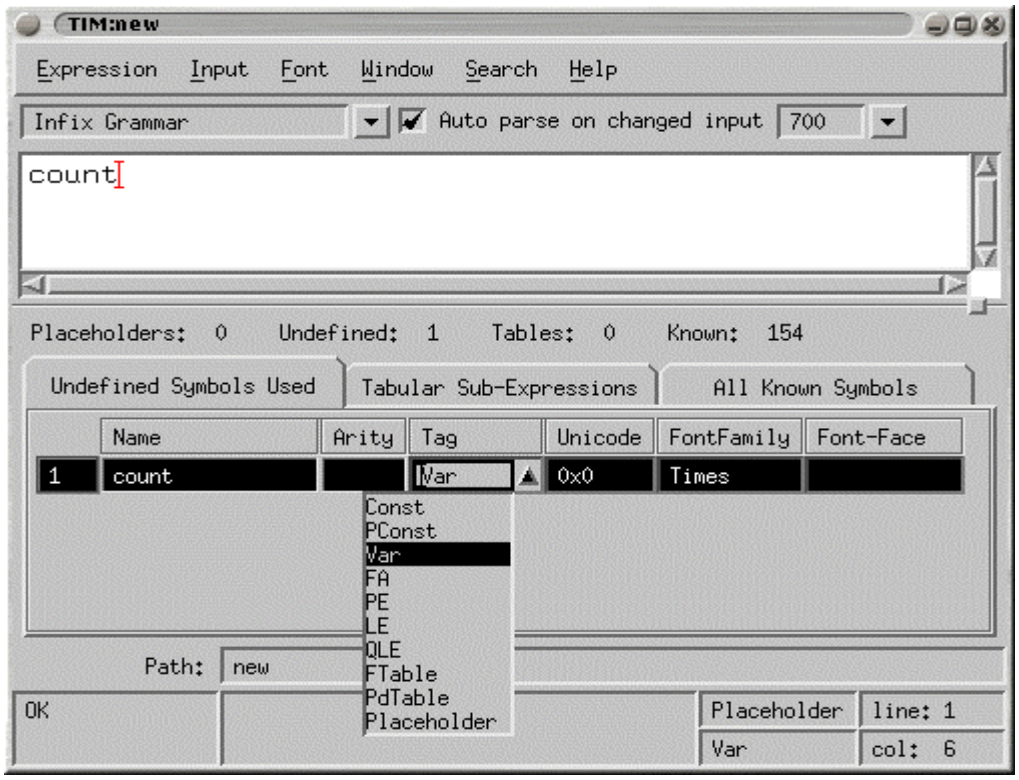


Figure 33: Editing new symbol properties

For each new symbol you can change a number of properties (see Table 12). To change the property click on the corresponding cell and modify its value.

The set of configurable properties, as well as their value ranges can be specified in the configuration file. For details see section *A.7 Configuration and Advanced features* below.

Property	Description	Value
Arity	number of arguments to a function, can be modified only when the Tag property is a function.	Integer value
Tag	Symbol type.	Select a list entry
Unicode	A single Unicode character representing this expression symbol. For example, symbol *universal* has the value of this property set as *0x2200*, which is the ∀ character.	Hexadecimal value
FontFamily	Font family used to display this symbol	
Font-Face	Font face used to display this symbol	

Table 12: New Symbol Properties

A.5.5.2 Launching TIM for Tabular Sub-Expressions

The expression that you're editing may also contain sub-expressions that are tables. These tabular sub-expressions are listed in the *Tabular Sub-Expressions* tab in the *Symbol Information Pages* area.

To edit a tabular sub-expression click on the *Tabular Sub-Expressions* tab, select a corresponding table entry in the list, press the right mouse button to popup a menu, and select *Edit Tabular Expn* menu item.

A.5.5.3 Modifying Definitions of Known Symbols

The properties of known symbols can be modified. To edit a known symbol, click on the *All Known Symbols* tab in the *Symbol Information Pages* area, scroll down to the symbol you want to edit, and click in the cell corresponding to the property you want to modify, and edit the value.

The set of properties that are displayed in the *All Known Symbols* tab can be modified in the configuration file (see section *A.7 Configuration and Advanced features* below).

A.5.6 Status Bar

The status bar (Figure 34) provides important information about the state of TIM. Each of the fields in the status bar contains important information you should be aware of. You can identify the status field by looking at the schematic in Figure 35. Field contents are summarised in Table 13.

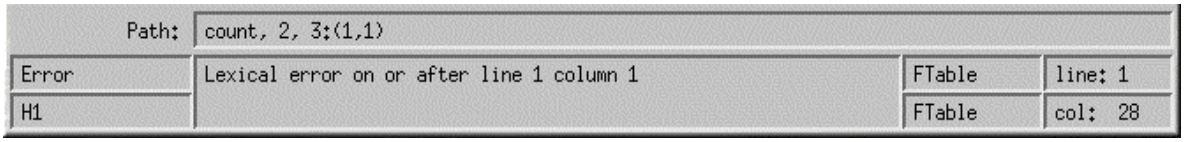


Figure 34: Status bar

	PATH		
STATUS	MESSAGE	ORIGINAL EXPN TYPE	LINE
GRID		CURRENT EXPN TYPE	COLUMN

Figure 35: Status bar fields

Field	Description of contents
PATH	The string representation of the path from root of the expression to the currently selected cell, or in case of non-tabular window the path to the entire expression.
STATUS	The input field status, which is one of: **Value** / **Description** OK — Input filed has been successfully parsed. Input Changed — Input field changed but not yet parsed. Parsing — Parsing the contents of the input field. Correcting — Correcting the contents of the input field. Error — Parsing of input field failed
GRID	Tracks the location of the mouse and indicates over which grid it's currently located.
MESSAGE	Parsing error message is displayed here.
ORIGINAL EXPN TYPE	The type of the original expression in the window, before modifications.
CURRENT EXPN TYPE	The type of the current expression in the window.
LINE	The line where the cursor is in the input area.
COLUMN	The column where the cursor is in the input area.

Table 13: Description of status bar field contents

A.5.7 Syntax Tree display tool

Some expressions can be complex and difficult to understand. To help you clarify the structure of the expression you can show the syntax tree for the expression in the input area by selecting **Help→Show Syntax Tree** from the menu. A dialog showing the syntax tree will be displayed (see Figure 36).

The Syntax Tree dialog will update automatically every time the parser is invoked. That is, if you have the auto-parse on and parsing was successful, the contents of the dialog will be updated to reflect the current syntax tree of the expression.

To further clarify the expression, you may choose to collapse some of the syntax tree sub-trees by double-clicking on a node – the child nodes will be hidden from the view. You can also click on the ⊟ icon. You can expand the hidden nodes by double clicking on them again (or by clicking on the ⊞ icon).

For each node (expression symbol) in the syntax tree the properties of symbol are also shown. To change which properties are to be displayed in the dialog see section *A.7 Configuration and Advanced features* below.

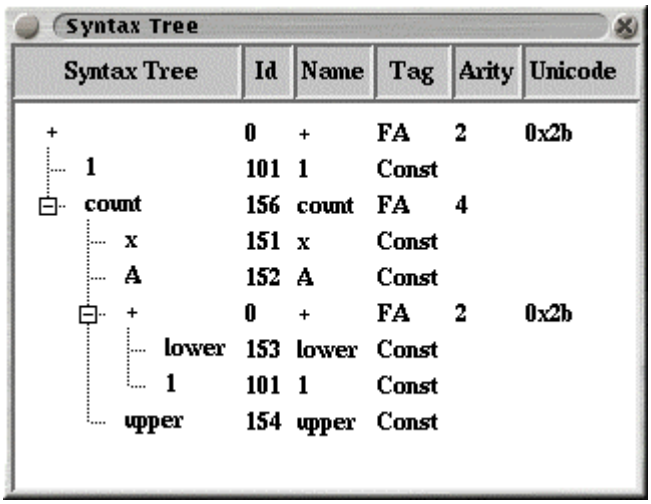


Figure 36: Syntax tree for expression 1+count(x, A, lower+1, upper)

A.6 Typing expression text

In TIM you enter the expression by placing the cursor in the input area and typing in the expression using the keyboard, or by pasting in text from X windows text buffer (middle mouse button action). Some of the expressions you may create will use mathematical symbols or arrows, such as ≥, Δ, ∀, →, and others. These special character symbols are not normally available on standard issue keyboard. You will need a special way of typing them in.

In order to create expressions using these special character symbols you must use a keymap. A keymap is a collection of predefined set of keystroke-sequence-to-character-symbol converters. That is, a sequence of keystrokes typed in the input area will result in a single special character symbol.

TIM comes with two standard keymaps, *TTS-input* and *TTS-Unicode*. *TTS-input* is the default keymap.

A.6.1 Typing a key sequence

To type in a non-keyboard symbol, you must use a predefined keyboard sequence. For example, when using the *TTS-input* keymap, to type in ∀ (universal symbol) you will type the following key sequence:

`uni<Esc>`

You will have noticed that as you type the sequence, it is being underscored, and as you type in the last key in the sequence, the entire underscored sequence of characters is being convert to the appropriate symbol (Figure 37).

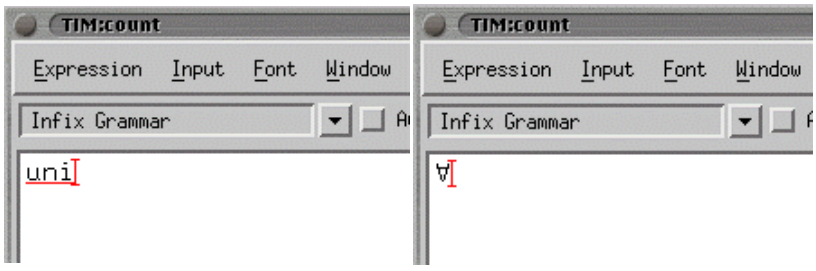


Figure 37: Typing a sequence for universal symbol and the resulting ∀ symbol

The *TTS-input* keymap defines key sequences for all the default symbols in the symbol table. For a complete list of key sequences see *Appendix E: TTS-input Keymap* below. The definitions of the TTS-input keymap keystroke sequences can be found in *${TTS_HOME}/tim/share/yudit/data/TTS-input.kmap*. To understand how to read the file, please read the *Defining a Custom Keymap* section below.

Other then the default *TTS-input* keymap, you may also choose to use the *TTS-Unicode* keymap. This keymap allows you to type in any symbol using its Unicode char value. For example, to input the ∀ symbol, you must type:

```
0x2200
```

Using this method you can type any Unicode symbol. *TIM-Unicode* keymap is very useful when the typed character does not have a defined key sequence. You can switch between keymaps at any time by selecting an appropriate entry from the **Input** menu.

A.7 Configuration and Advanced features

TIM can be configured in multiple ways. The user can modify the keyboard mappings, add input parsers, change display fonts, add new table templates, add specify valid values for new symbol properties (info classes), control which symbols properties are displayed in the symbol info pages area, and other.

All TIM configuration is in *.TTS-timrc* file found in your home directory. It is divided in to sections. Each section of the file has a specific purpose (see Table 14).

Section	Description
Default	Default setting for various items (Yudit)
Menu Item	Menu Items (Yudit)
TIM TTS Fonts	Map FontFamily and Font-face combinations to X-font
TIM Java Parsers	Specify where to find input parsers
TIM Table Templates	Table type configurations, such as location and dimensions of grids
TIM Unicode Map	Unicode to Motif Strings mappings
TIM Info Display Format	Display formats for symbol properties (info classes)
TIM Editor Misc	Various TIM specific preferences

Table 14: TIM configuration file sections

A.7.1 Defining a Custom Keymap

You can customize how to type in Unicode characters by constructing a custom keymap file. You can define any number of unique keymap files, and switch them at any time. Each keymap file contains a list of keymap definitions. A keymap definition is composed of key sequence and the resulting Unicode character.

A key sequence in the keymap definition must be unique – if two definitions use the same key sequence, the last definition will prevail. However, there may be multiple definitions resulting in the same Unicode character.

To create a custom keymap, you must:

- Create custom keymap file that contains the keystroke definitions
- Modify the TIM configuration file to indicate the name and location of the new keymap.

A.7.1.1 Keymap file syntax

Keymap files are text files with *.kmap* extension and contain keystroke definitions. The syntax of the keymap file is described in Table 15.

Syntax	Description
// Comment	Comments start with a pair of '/' and continue to the end of the line
"key-sequence = uchar",	Specifies a keystroke definition. The left side of the equation indicates a key sequence and right side the Unicode char value (uchar) the sequence converts to. Note the surrounding pair of *double-quote* and terminating *comma* characters.

Table 15: Keymap file syntax

The key sequence is composed of either ASCII characters or hexadecimal code representing a special key. Special keys are summarised in Table 16.

Hex Code	Key	Hex Code	Key	Hex Code	Key
0x1b	**<Esc>**	0x3d	**<=>**	0x0D	**<Ctrl-m>**
0x20	**< >** (Space)	0x78	**<x>**	0x0E	**<Ctrl-n>**
0x22	**<”>**	0x01	**<Ctrl-a>**	0x10	**<Ctrl-o>**
0x2C	**<,>**	0x02	**<Ctrl-b>**	0x11	**<Ctrl-p>**
0x30	**<0>**	0x03	**<Ctrl-c>**	0x12	**<Ctrl-q>**
0x31	**<1>**	0x04	**<Ctrl-d>**	0x13	**<Ctrl-r>**
0x32	**<2>**	0x05	**<Ctrl-e>**	0x14	**<Ctrl-s>**
0x33	**<3>**	0x06	**<Ctrl-f>**	0x15	**<Ctrl-u>**
0x34	**<4>**	0x07	**<Ctrl-g>**	0x16	**<Ctrl-v>**
0x35	**<5>**	0x08	**<Ctrl-h>**	0x17	**<Ctrl-w>**
0x36	**<6>**	0x09	**<Ctrl-i>**	0x18	**<Ctrl-x>**
0x37	**<7>**	0x0A	**<Ctrl-j>**	0x19	**<Ctrl-y>**
0x38	**<8>**	0x0B	**<Ctrl-k>**	0x1A	**<Ctrl-z>**
0x39	**<9>**	0x0C	**<Ctrl-l>**		

Table 16: Keymap file - special key hex codes

The user can also use keyboard function keys to define key mappings. Function keys require special sequences summarised in Table 17. An example of a keymap file is shown in Figure 38.

Sequence	Key	Sequence	Key	Sequence	Key
<F1> 0x1b	**<F1>**	<F5> 0x1b	**<F5>**	<F9> 0x1b	**<F9>**
<F2> 0x1b	**<F2>**	<F6> 0x1b	**<F6>**	<F10> 0x1b	**<F10>**
<F3> 0x1b	**<F3>**	<F7> 0x1b	**<F7>**	<F11> 0x1b	**<F11>**
<F4> 0x1b	**<F4>**	<F8> 0x1b	**<F8>**	<F12> 0x1b	**<F12>**

Table 17: Keymap file - special key sequences for function keys

```
// Nice-input.kmap
//
// This is the file corresponding to the Nice-input keymap indicated
// in the TTS-timrc configuration file, and it contains private
// keystroke mappings for Table Tool System's Table Input Method.

"arrowboth 0x1b  = 0x2194", // Arrow Both
"<->             = 0x2194", // Arrow Both (alternate mapping)

"arrowdown 0x1b  = 0x2193", // Arrow Down
"0x04            = 0x2193", // Arrow Down (alternate mapping)

"universal 0x1b  = 0x2200", // Universal / For All
"<F1> 0x1b       = 0x2200", // Universal / For All (alternate mapping)
```

Figure 38: Keymap file example

A.7.1.2 Configuration file changes

Once you have created a new keymap file, you must specify where to find the key sequences file. To do this you must edit *.TTS-timrc,* the TIM configuration file, located in your home directory.

This file is divided into sections separated by the section name in a pair of square brackets. For example, the *default section* starts with a line that contains:

```
[Default]
```

Each section contains a set of variables. The same variable names may be used in different sections. Such variables are considered to be distinct, and each has a different meaning.

The relevant sections and variables for specifying your own keyboard are summarized in

Section	Variable	Description
[Default]	Path	Defines search path where to locate the keymap file
[Default]	Input	Defines the default keymap
[Menu Item]	Input	Specifies the list of permissible keymaps

Figure 39: Sections and variables needed for configuring custom keymap

To add the name of the keymap you have just created you must update the values of variables specified in Figure 39. Figure 40 is an example showing how to make the *Nice-input* keymap defined in file *${HOME}/mytts/Nice-input.kmap* as the default keymap.

```
[Default]
Path=${HOME}/mytts
Input=Nice-input
...
[Menu Item]
...
Input=TTS-input,TTS-Unicode,Nice-input
```

Figure 40: Adding the Nice-input keymap to TIM configuration file

A.7.2 Changing TTS fonts

Expression symbols use FontFamily and Font-Face properties as distinguishing features. These two properties govern what font is used to render them. This means that two symbols may be very similar, except that they use different fonts for display.

TIM uses these properties to render the expressions in the cells of the table. Each symbol is rendered using the font information. The font information is composed of FontFamily and Font-Face properties. The value of FontFamily might be “Times”, “Helvetica”, or other. The value of Font-Face is a number – the number meanings are listed in Table 18. However, TIM must know what X font to use to render the symbol. To do this TIM maps symbol’s font info to an X font.

Value	Description
0	Normal
1	Italic
2	Bold
3	Italic-Bold

Table 18: Valid Font-Face values

A mapping consists of a tag and a X-font name. Composing the FontFamily and Font-Face symbol info values with a *hyphen* creates a mapping tag. For example, a tag for "Times" FontFamily and "0" Font-Face is "Times-0". Each tag is assigned an X font. For example, a mapping for tag "Times-0" might be as follows:

```
Times-0     = -*-times-*-r-*--*-120-*
```

To specify a complete set of mappings in the configuration file, you must first list all the tags in the *Fonts* variable. The font tag mappings must follow.

Finally, the configuration file must always specify a *DefaultFont* mapping. TIM will use this font to render any symbol that does not have a valid font mapping.

```
[TIM TTS Fonts]

DefaultFont = -*-times-*-r-*--*-120-*

## List of the font tags
Fonts=Times-0,Times-1,Times-2,Times-3,Symbol-0

## Mapping tags to X-fonts
Times-0     = -*-times-*-r-*--*-120-*
Times-1     = -*-times-*-i-*--*-120-*
Times-2     = -*-times-bold-r-*--*-120-*
Times-3     = -*-times-bold-i-*--*-120-*
Symbol-0    = -*-symbol-*-*-*-*-120-*
```

Table 19: Sample *TIM TTS Fonts* configuration file section

A.7.3 Specifying New Table Templates

TIM renders tables according to table templates. Each table has the following properties:

Property	Description
Type	
ccgCase	The template's CCG case.
Structure	
container_dimension	Number of lines (rows of grids) and stacks (columns of grids).
grid_placement	Placement of grids in the container
grid_dimensions	Maximal dimensions of the grids in the container
grid_lines_min_rows	Minimum number of rows for each grid line
grid_lines_max_rows	Maximum number of rows for each grid line
grid_stacks_min_columns	Minimum number of columns for each grid stack
grid_stacks_max_columns	Maximum number of columns for each grid stack
Display	
grid_lines_min_heights	Minimum display height for each a grid line in number of pixels
grid_lines_heights	Default display height for each grid line in number of pixels.
grid_stacks_min_widths	Minimum display width for each a grid stack in number of pixels
grid_stacks_widths	Default display width for each grid stack in number of pixels

Table 20: table template properties

A.7.3.1 Type properties

Each table symbol has a CCG case. To render a table you can use only the templates that mach the table's CCG case. To specify the template's CCG case set the ccgCase variable. Valid CCG values are listed in Table 21.

Value	Description
0	Normal
1	Inverted
2	Vector
3	Decision

Table 21: Valid CCG values.

A.7.3.2 Structural properties

The structural properties govern the placement of grids, maximum permissible dimensions of grids maximum and minimum number of rows and columns in each grid.

The `container_dimension` property governs how many lines and stacks of grids there are in the template. The stacks and lines create a matrix. A grid can only be placed at the intersection of the line and a stack.

The `grid_placement` property specifies what grid to place at what cell in the matrix. The values that can be placed at the stack and line intersections in the matrix are *G, H1, …, Hk* for the table grids, or *E* for no-grid.

The `grid_dimensions` property specifies what is the maximum grid dimension for each metric cell. Currently, only values *1* or *2* are allowed for grids *G, H1, …, Hk,* and *0* for *E* (no-grid).

TIM assumes that grids that are next to each other top-to-bottom must have the same number of columns, and the grids that are next to each other left-to-right have the same number of rows. For example, in Figure 41 the following must hold true: X=Y and I=J.

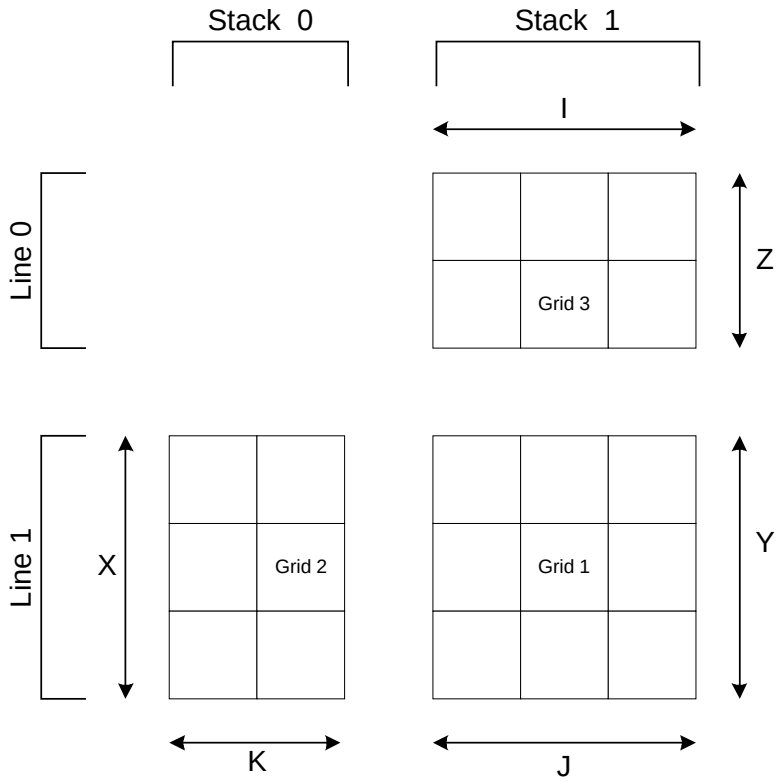


Figure 41: Matching row and column count for neighbouring grids

TIM enforces this assumption by ensuring that 1) when a table is created the neighbouring grids have coinciding number of rows or columns, and 2) rows (columns) are inserted or deleted in all the grids in the line (stack) at the same time.

The number of rows and columns is further constrained by `grid_lines_min_rows`, `grid_lines_max_rows`, `grid_stacks_min_columns`, `grid_stacks_max_columns` properties. The first two properties govern the min and max number of rows per line of grids, and the last two properties govern the min and max number of columns per stack of grids.

For example, suppose that table in the Figure 41 uses a template that specifies `grid_lines_min_rows` to be {1, 3}, and `grid_stacks_min_columns` to be {2, 3}. Then, we know that Grid 2 cannot have less than 3 rows and 2 columns.

The template display properties govern what are the minimum and maximum grid dimensions when displayed on the screen. These properties are independent of the row and column values – they only govern what space is to be used for the entire grid. If a grid does not have enough screen space to display all the rows or columns it contains the user will need to use the scroll bars to scroll the grids as needed.

```
##############################################################################
## Normal Table - Up-Down case
##############################################################################
Normal-LeftRight.ccgCase = 0
Normal-LeftRight.container_dimension = 2, 2
Normal-LeftRight.grid_placement = \
   E, H1; \
  H2,  G;
Normal-LeftRight.grid_dimensions = \
   0,  1; \
   1,  2;

Normal-LeftRight.grid_lines_min_rows =     1, 1
Normal-LeftRight.grid_lines_max_rows =     1, MAX
Normal-LeftRight.grid_stacks_min_columns = 1, 1,
Normal-LeftRight.grid_stacks_max_columns = 1, MAX

Normal-LeftRight.grid_lines_min_heights =  0,  80
Normal-LeftRight.grid_lines_heights     = 80, 110

Normal-LeftRight.grid_stacks_min_widths =   0, 150
Normal-LeftRight.grid_stacks_widths     = 130, 200
```

Figure 42: Template configuration example

Figure 42 show an example of a template. The template uses is a 2-by-2 grid matrix, with upper left cell empty. Header grids have maximum 1 dimension, the main

grid has maximum 2 dimensions. Header grid H1 must have 1 row, and header grid H2 must have 1 column – no more, no less. Grid G must have at least one row and one column.

All templates must be listed in `TableTemplates` variable. Each template definition must follow.

A.7.4 Unicode Character Mapping

Some Unicode characters can be found in either a Latin-1 or Symbol font. You can configure the mapping in the *TIM Unicode Map* section of the configuration file. In this section you must specify the values for two variables: *DefMap* and *Ranges*.

The *DefMap* variable specifies default Unicode mapping. If a Unicode character does not have a mapping, it will be mapped to this default value. The `¿' is a good choice (Latin-1 character 0xbf).

Since there are quite a few Unicode characters and we don't want to map them all, the *Ranges* variable indicates what Unicode character ranges we are interested in. We really care only for some of the Letter-Like Symbols (0x2100-0x214f), Arrows (0x2190-0x21ff) and Mathematical Operators (0x2200-0x22ff). Note that you should not specify the Latin-1 set (0x0000-0x00FF) since this is already done for you by default.

Finally, you can define a mapping for every Unicode symbol you need. Each mapping is composed of a "`Unicode=FontID CharID`" entry, for example:

```
0x2200=0x02 0x22
```

The Unicode value is a 4-digid hexadecimal number, FontID and Char ID values 2 digit hexadecimal numbers. Valid values for FontID are 0x01 for Latin-1 font and 0x02 for Symbol font.

In the above example the Unicode character 0x2200 (∀) is mapped to font 0x02 (Symbol) character 0x22 (∀). A sample *TIM Unicode Map* section is shown in Figure 43.

```
[TIM Unicode Map]

DefMap=0x01 0xbf
Ranges=0x2200-0x22f1

## For All
0x2200=0x02 0x22
## Partial Differential
0x2202=0x02 0xb6
## There Exists
0x2203=0x02 0x24
...
```

Figure 43: Sample TIM Unicode Map configuration file section

A.7.5 Symbol Property Type (Info Class) Display Formats

The symbol property type (info class) can be at times cryptic. For example, Font-Face value is number between 0 and 3 (Table 18) – any other number is invalid. To help you better understand and remember what the valid values are, TIM allows you to specify the valid value ranges and how values are displayed. This is done in the *TIM Info Display Format* section.

All the names of the info classes are listed, in sequence, in the *InfoClasses* variable. Then, for every info class listed you must specify the *type, numChar* and optionally *enumVal* attributes, described in Table 22.

The info class attribute information is used to control the display of the symbol properties in the *symbol info pages*. The *type* attribute governs what kind of editor to use to edit the property (Table 23), and the *numChar* specifies the width of the column showing the given info class.

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>Type</td><td>The type of the info class:
<table><tr><th>Value</th><th>Meaning</th></tr><tr><td>String</td><td>Latin-1 string</td></tr><tr><td>Binary</td><td>Binary data</td></tr><tr><td>decimal</td><td>Decimal integer value</td></tr><tr><td>hexadecimal</td><td>Hexadecimal integer value</td></tr><tr><td>Enum</td><td>Enumerated value list</td></tr></table></td></tr>
<tr><td>numChar</td><td>Number of characters shown in the display field</td></tr>
<tr><td>enumVal</td><td>If type attribute is enum, a comma separated list of values</td></tr>
</table>

Table 22: Info class format attributes

Type	Editor
string	A text editor that accepts any Latin-1 string.
binary	*Not supported*
decimal	A text editor that accepts only decimal values.
hexadecimal	A text editor that accepts only hexadecimal values.
enum	A combo box that lists values specified in the *enumVal* and allows a selection of only one value from this list.

Table 23: Info class editor by type.

For example, the attributes of *Tag* info class are defined as follows:

```
Tag.type    = enum
Tag.numChar = 9
Tag.enumVal = Const, PConst, Var, FA, PE, LE, QLE, FTable, \
              PdTable, Placeholder
```

These attribute settings result in a combo box being used to edit the value of the symbol property, and a display field 9 characters wide (Tag column):

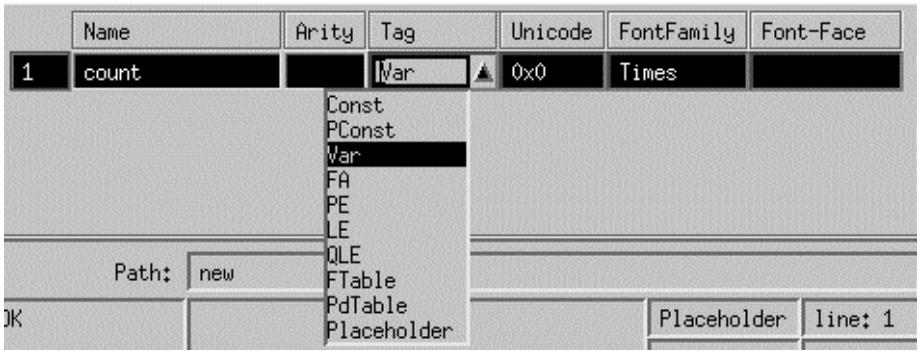


Figure 44: Editor for an enum type info class (Tag)

Here is a sample *TIM Info Display Format* section of the configuration file:

```
[TIM Info Display Format]
InfoClasses = Name, FontFamily, Font-Face

Name.type          = string
Name.numChar       = 16

FontFamily.type    = string
FontFamily.numChar = 10
FontFamily.strList = Times, Helvetica, Courier, Symbol

Font-Face.type     = enum
Font-Face.numChar  = 11
Font-Face.enumVal  = Normal, Italic, Bold, Italic-Bold
```

Figure 45: Sample *TIM Info Display Format* configuration file section

A.7.6 Selecting Symbol Info Classes to Display

A number of TIM components can be configured to display only certain properties of a symbol in the *TIM Editor Misc* section of the configuration file.

The components and the variables that govern the how the info classes are displayed are summarised in Table 24. A sample section info class display configuration is shown Figure 46.

Variable	Description
Undefined Symbols List	
`UndefinedSymbolsShowInfoClasses`	Info classes (property types) to show for each undefined symbol in the list
`UndefinedSymbolsRequiredValues`	Properties that must be defined for every symbol class
`UndefinedSymbolsNonEditable`	Properties that are not editable
All Known Symbols List	
`KnownSymbolsInfoClasses`	Info classes (properties) to show for every symbol in the list
`KnownSymbolsNonEditable`	Properties that are not editable.
Tabular Sub-Expression List	
`TableSymbolShowInfoClasses`	Info classes to show for symbols corresponding to the tabular sub-expressions
Syntax Tree Display Dialog	
`SyntaxTreeShowInfoClasses`	Info classes to show in the Syntax Tree display dialog.

Table 24: Variables used for component info class symbol display configuration

```
 [TIM Editor Misc]

UndefinedSymbolsShowInfoClasses = Name, Arity, Tag, Unicode, \
                                  FontFamily, Font-Face
UndefinedSymbolsRequiredValues  = Tag, FontFamily, Font-Face
UndefinedSymbolsNonEditable     = Name, Arity
KnownSymbolsInfoClasses         = Name, Arity, FontFamily, Font-Face,
                                  Tag, CCG, Unicode
KnownSymbolsNonEditable         =
TableSymbolShowInfoClasses      = Name, Tag, CCG
SyntaxTreeShowInfoClasses       = Name, Tag, Arity, Unicode
```

Figure 46: Sample configuration for which info classes to display

A.7.7 Auto-Parse Configuration

The following table lists variables that configure auto parse behaviour:

Variable	Description
`AutoParseOn`	Determines if auto-parse is on by default. Specify *Yes* or *No.*
`AutoParseDelayList`	A comma separated list of integer values specifying auto-parse time delays in milliseconds. The time delay is the time to wait after last input area change before engaging the parser.
`AutoParseDelayListDefaultIndex`	Index into the `AutoParseDelayList` indicating the default auto-parse time delay.
`MaxAutoCorrections`	Number of auto-corrections the parser will make. A value of 0 indicates no auto corrections at all, including no injections of the placeholder symbol "?".

Figure 47: Auto-parse configuration variables

Sample configuration:

```
AutoParseOn = No
AutoParseDelayList = 500, 700, 1000, 2000, 4000
AutoParseDelayListDefaultIndex = 1
MaxAutoCorrections = 8
```

Figure 48: Sample auto-parse configuration

Appendix B: Writing a Custom TTS Parser

To create an expression you type in the expression string in the input area. The string is then parsed and a syntax tree for the input string is computed. Using the syntax tree TIM matches the nodes in the tree to the known symbols in the symbol table.

To find the syntax tree node's symbol we seek a symbol that can match it Name or Unicode property to the node's string, and then we check if the Arity property also matches the node's arity (child count).

If such symbol is found, we mark the node with the symbol id. If the symbol is not found (symbol is undefined), we create a new symbol with the desired properties and display this symbol in the *Undefined Symbols* list in the Symbols Information Area.

When the user defines an undefined symbol the symbol is moved to the *Known Symbols* list with the properties specified by the user.

Therefore, to create an expression in TIM a syntax-tree producing parser is needed. To facilitate creation of input parsers TIM comes with a set of parser-generating tools based on JavaCC - a popular Java Compiler Compiler.

The TIM parser generator takes a grammar definition file and produces a Java-based lexicographic analyzer (lexer) and parser. The generated parsers are an LL(k) parsers. That is, when processing the input the parser uses k look-ahead tokens to determine which rule to apply next. [22] is good reference describing how LL(k) parsers work.

TIM contains a framework that supports multiple parsers, so long as the parsers are generated parsers and conform to the same API. TIM grammar definition files have extension "*.tg*".

B.1 Parser Specification File Format

The parser specification file is divided into four sections: options, parser class declaration, token declarations, and parser rules.

Section	Description
Options	JavaCC directives (*do not change*)
Class declaration	Declares the parser class name (*do not change*)
Scanner declarations	Lexer specification
Parser productions	An LL(k) parser specification

Table 25: Parser file sections

B.1.1 Scanner Declarations

The anatomy of the regular token declarations is defined as follows:

```
TOKEN : {
< [#]TokenName : RegularExpression >
|  < [#]TokenName : RegularExpression >
|  < [#]TokenName : RegularExpression >
|  ...
}
```

TOKEN: { … } declares a token grouping – there may be several such constructs in a file. *TokenName* is specifies the name of the token. A "*#*" symbol declares the token as *private*. Private tokens can only be used within the current as part of the regular expressions that define other tokens. *RegularExprssion* is the regular expression used to match the input text and mark it as the token indicated by TokenName. The "|" is token declaration separator – there may be several tokens declared within the current token grouping.

You can also declare what input to skip without creating tokens. Skip is defined as follows:

```
SKIP : {
< RegularExpression >
|  < RegularExpression >
|  < RegularExpression >
|  ...
}
```

For example, the following declares to skip all white space and match integer values as INGEGER_LITERAL tokens:

```
SKIP :
{
  " " | "\t" | "\n" | "\r"
}

TOKEN :
{
  <INTEGER_LITERAL:
        <DECIMAL_LITERAL>     (["l","L"])?
         | <HEX_LITERAL>      (["l","L"])?
         | <OCTAL_LITERAL>    (["l","L"])? >
 | <#DECIMAL_LITERAL:         ["1"-"9"] (["0"-"9"])* >
 | <#HEX_LITERAL:       "0"["x","X"](["0"-"9","a"-"f","A"-"F"])+>
 | <#OCTAL_LITERAL:     "0" (["0"-"7"])* >
}
```

B.1.2 Parser Productions

Lets do a quick introduction to parsing. First, definitions of components of a parser:

1. A finite set of *terminal vocabulary* V_t – this is the token set produced by the scanner.
2. A finite set of different, intermediate symbols, called the *non-terminal vocabulary* V_n.
3. A *goal symbol* (a non-terminal)
4. A finite set of productions (or rules) of the form $A \rightarrow Y_1 \ldots Y_m$, where

   $A \in V_n$

   Y_i is of the form $X_{i,1} \ldots X_{i,n}$ or $(X_{i,1} \ldots X_{i,n})^*$ or $(X_{i,1} \ldots X_{i,n})^+$ or $(X_{i,1} \ldots X_{i,n})?$

   $Xij \in Vn \cup Vt$

   $1 \leq i \leq m, m \geq 0$

   $1 \leq j \leq n, n \geq 0$

   Where * means zero or more repetitions, + means one or more repetition, and ? means zero or 1 repetition.

The syntax of the TIM parser productions then is as follows:

Construct	Description
<TokenName>	A token, member of V_t. Token TokenName must be declared in the scanner section.
NonTerminal()	A non-terminal, member of V_n

`void` *`NonTerminal()`*`: #Node {}`
`{`
$Y_1 \ldots Y_m$
`}`
	Declares a NonTerminal$\rightarrow Y_1 \ldots Y_m$ rule. #NodeName declares that a syntax tree will always be constructed for this rule.

// comment	Comment starts at // until the end of line
/* comment */	Comment is contained between matching /* and */
LOOKAHEAD(k)	Declares to use local look-ahead of k tokens. Used for rule conflict resolution.
@TTS_NODE@	Declares the preceding token to be a representative token for the rule – this information is used to reconstruct input string from the stored syntax tree (see Appendix C: Translating Stored Expressions to Strings).
($Y_1 \ldots Y_m$) #ConditionalNode(>n)	This optional construct placed around $Y_1 \ldots Y_m$ indicates that a syntax tree node will be constructed only if as a result of processing of non-terminals it will have more than *n* child nodes. If there are less than *n* child nodes, node is not created. The child nodes will be assigned to a nearest node up the tree.

Table 26: Meaning of parser rule constructs

The rules of a TIM parser are *further constrained* by the input string reconstruction algorithm described in Appendix C: Translating Stored Expressions to Strings below.

```
//
// Start rule - always present do not change!
//
Node Start() #Start : {}
{
      Expression()
      { return jjtThis; }
}

////////////////////////////////////////////////////////////////
// rules
////////////////////////////////////////////////////////////////
void Expression() : {}
{
      ExistentialFormula()

      // DefinedFunction
      | DefinedFunction()

}

void ExistentialFormula() : {}
{
      (
            LOOKAHEAD(2)
            (
                  "("
                  <EXISTS>                                  @TTS_NODE@
                  (Identifier() | Placeholder())
                  <SEPARATOR>
                  UniversalFormula()
                  ")"
            )
            | UniversalFormula()
      ) #ExistentialFormula(>1)
}
```

Figure 49: Sample of TIM parser rules

To compile the .tg file containing the TIM input parser definition, perform the following sequence of commands :

```
java -classpath $(TTS_ROOT) tim.generator.ParserSupportGenerator Parser.tg
java -classpath $(TTS_ROOT) tim.generator.RuleBookGenerator Parser.jjt
$(TTS_ROOT)/javacc/bin/jjtree Parser.jjt
$(TTS_ROOT)/javacc/bin/javacc Parser.jj
$(TTS_ROOT)/javacc/bin/jjdoc Parser.jj
javac *.java
```

Note that Java 2 is required to compile and run the TIM parsers.

B.2 Custom Error Correction

When a TIM parser specification file is compiled a default error recovery utility is also created. This utility is a Java source file that gets compiled along with the rest of the parser source files. You can change the behaviour of the error recovery utility by modifying the appropriate error recovery Java source file.

Assuming the parser specification is located in *MyParserSpecification.tg*, then the name of the error recovery source file is *YourParserSourceFileErrorRecovery.java.*

B.3 Specifying Location of Parsers

Location of parsers can be specified in the configuration file in the *TIM Java Parsers* section. In this section you must define three variables *CLASSPATH*, *Parsers* and *ParserDefaultIndex*. The *CLASSPATH* variable defines the class path to be used to locate the Java class files for the input parser. The *Parsers* variable specifies a list of parser tags. The *ParserDefaultIndex* indicates which of the parsers specified in *Parsers* is the default parser.

Parser tags are defined below the *Parsers* variable. Each tag is assigned two comma-separated values. The first value is the name of the parser (shown in the Notation Selector), the second is the parser class name. A sample *TIM Java Parser* section of the configuration file is shown in Figure 50.

```
[TIM Java Parsers]

##
## Classpath in addition to the default ${TTS_HOME}
##
CLASSPATH=

Parsers = parser1, parser2
#Parsers=parser2,parser1

parser1 = Infix Grammar, tim.grammars.g1.TTSGrammarParser
parser2 = Prefix Grammar, tim.grammars.g2.PrefixParser

ParserDefaultIndex = 0
```

Figure 50: Sample input parser configuration

Appendix C: Translating Stored Expressions to Strings

TIM is capable to reconstruct input text that represents the stored syntax tree of an expression. That is, this constructed string will result when parsed in exactly the tree that has been stored.

First, let's define an *expansion* to be the unrolling of a rule's right-hand side for so that it contains a specified number of non-terminals. Expansion converts the *, + and ? repetitions into concrete number of non-terminals.

An expansion of the rule involves only the symbols in the rule, and does not follow the rules of the non-terminals' to generate the desired non-terminal count.

For example, suppose the following grammar:

```
A ::= (f B | g C)*
B ::= b D
C ::= c E
D ::= d
E ::= e
```

Here the upper case letters are non-terminals and lower case letters are tokens. Suppose you want an expansion of A with two non-terminals, then *expansion(A,2)* gives only: f B f B, f B g C, g C f B, g C g C.

The input string reconstruction algorithm, which we will present shortly, makes some basic assumptions made about the parser rules:

1. All node-producing rules must produce expansions that contain *exactly* one token marked with @TTS_NODE@ (known as the *node token*) each. An expansion may have any number of non-node tokens called *punctuation tokens*.

2. When examining a node in the syntax tree and matching the node's symbol to a node producing rule, the rule must expand to contain *n* non-terminals, where *n* is exactly the number of child nodes

3. Rules that do not produce nodes do not have any node tokens. They may have any number of punctuation nodes.

4. Rules intended for producing leaf nodes in the syntax tree will use a single token. This token must be the node token.

These rules must be adhered to when constructing TIM input parser.

We are now ready to present the parser-driven expression input string reconstruction algorithm. The algorithm is as follows (Figure 51):

```
Start with root node r
   Do {
      Let s be the node's symbol, and n be the number of children. Then,
      find a node-constructing rule r in the grammar that:
             1.   Symbol s is one of the node tokens defined for the
             rule
             2.   Expansion set E = expansion(r, n, s), E ≠ ∅

      For every e ∈ E do {
         Recursively find rules for non-terminal T_i ∈ e such that the
         current node's child_i symbol is the node token of R.  If no
         such node was found, try the next expansion.  If such node is
         found, create a string that is a collection all the punctuation
         tokens encountered while recursively seeking R combine it with
         s in right order, and return the string.
      }
   }
}
```

Figure 51: Parser-driven algorithm for expression input string reconstruction

Appendix D: TTS modifications to support TIM

D.1 Unicode – the new symbol class

A new symbol class was added to indicate what Unicode character to use to represent the symbol.

D.2 Introduction of Placeholder symbol Tag

A new Placeholder tag was added to the TTS kernel. Previously, placeholder symbol was a specific symbol in the table, but this made processing of expressions complicated because it made this one symbol an exception to the rule.

An introduction of Placeholder tag made the design much cleaner. It also simplifies creation of multiple kinds of placeholder symbols – something that might be used in the future development.

D.3 C++ support – modifications to TTS tool public header files

To resolve linking errors, most of the kernel header files had modified so that C++ classes could correctly use C kernel utilities. This did not affect the kernel, since the C++ changes are wrapped with #ifdef statements.

Appendix E: TTS-input Keymap

```
// TTS-input.kmap for the Unicode editor Yudit
// Yarek J. Kowalik

// Keystroke mappings for Table Tool System's Table Input Method
// non-ASCII default symbols.

// Note 0x1b is ESC

"ab 0x1b   = 0x2194", // Arrow Both
"ab!       = 0x2194", // Arrow Both

"ad 0x1b   = 0x2193", // Arrow Down
"ad!       = 0x2193", // Arrow Down

"al 0x1b   = 0x2190", // Arrow Left
"al!       = 0x2190", // Arrow Left

"ar 0x1b   = 0x2192", // Arrow Right
"ar!       = 0x2192", // Arrow Right

"au 0x1b   = 0x2191", // Arrow Up
"au!       = 0x2191", // Arrow Up

"db 0x1b   = 0x21d4", // Double Arrow Both
"db!       = 0x21d4", // Double Arrow Both
"<F2> 0x1b = 0x21d4", // Double Arrow Both

"dd 0x1b   = 0x21d3", // Double Arrow Down
"dd!       = 0x21d3", // Double Arrow Down

"dl 0x1b   = 0x21d0", // Double Arrow Left
"dl!       = 0x21d0", // Double Arrow Left

"dr 0x1b   = 0x21d2", // Double Arrow Right
"dr!       = 0x21d2", // Double Arrow Right
"<F1> 0x1b = 0x21d2", // Double Arrow Right

"du 0x1b   = 0x21d1", // Double Arrow Up
"du!       = 0x21d1", // Double Arrow Up

"& 0x1b    = 0x2227", // Logical And
"and 0x1b  = 0x2227", // Logical And
"and!      = 0x2227", // Logical And
"&&        = 0x2227", // Logical And

"! 0x1b    = 0x00ac", // Logical Not
"not 0x1b  = 0x00ac", // Logical Not
"not!      = 0x00ac", // Logical Not
"!!        = 0x00ac", // Logical Not

"| 0x1b    = 0x2228", // Logical Or
"or 0x1b   = 0x2228", // Logical Or
"or!       = 0x2228", // Logical Or
"||        = 0x2228", // Logical Or
```

```
". 0x1b    = 0x00b7", // Bullet
"bul 0x1b  = 0x00b7", // Bullet
"bul!      = 0x00b7", // Bullet

"0x01      = 0x2200", // Universal / For All
"A!        = 0x2200", // Universal / For All
"a 0x1b    = 0x2200", // Universal / For All
"a!        = 0x2200", // Universal / For All
"uni!      = 0x2200", // Universal / For All
"uni 0x1b  = 0x2200", // Universal / For All

"0x05      = 0x2203", // Existential
"E!        = 0x2203", // Existential
"e 0x1b    = 0x2203", // Existential
"e!        = 0x2203", // Existential
"exi 0x1b  = 0x2203", // Existential
"exi!      = 0x2203", // Existential

"ns 0x1b   = 0x2284", // Not Subset
"ns!       = 0x2284", // Not Subset

"nel 0x1b  = 0x2209", // Not Element
"nel!      = 0x2209", // Not Element

"! 0x3d    = 0x2260", // Not Equal
"neq 0x1b  = 0x2260", // Not Equal
"neq!      = 0x2260", // Not Equal

"el 0x1b   = 0x2208", // Element
"el!       = 0x2208", // Element

"psub 0x1b = 0x2282", // Proper Subset
"psub!     = 0x2282", // Proper Subset

"psup 0x1b = 0x2283", // Proper Superset
"psup!     = 0x2283", // Proper Superset

"equiv 0x1b= 0x2261", // Equivalence
"equiv!    = 0x2261", // Equivalence
"0x3d 0x3d = 0x2261", // Equivalence (0x3d is ‘=‘ ie. == converts to
Equivalence)

"rsub 0x1b = 0x2286", // Reflex Subset
"rsub!     = 0x2286", // Reflex Subset

"rsup 0x1b = 0x2287", // Reflex Superset
"rsup!     = 0x2287", // Reflex Superset

"< 0x3d    = 0x2264", // Less or Equal
"le 0x1b   = 0x2264", // Less or Equal
"le!       = 0x2264", // Less or Equal

"> 0x3d    = 0x2265", // Greater or Equal
"ge 0x1b   = 0x2265", // Greater or Equal
"ge!       = 0x2265", // Greater or Equal

"u 0x1b    = 0x222a", // Union
"u!        = 0x222a", // Union
"union 0x1b= 0x222a", // Union
"union!    = 0x222a", // Union

"i 0x1b    = 0x2229", // Intersection
"i!        = 0x2229", // Intersection
"inter 0x1b= 0x2229", // Intersection
"inter!    = 0x2229", // Intersection
```

```
"* 0x1b    = 0x00d7", // Multiply
"*!        = 0x00d7", // Multiply
"mul 0x1b  = 0x00d7", // Multiply
"mul!      = 0x00d7", // Multiply

"/ 0x1b    = 0x00f7", // Divide
"/!        = 0x00f7", // Divide
"div 0x1b  = 0x00f7", // Divide
"div!      = 0x00f7", // Divide

"+-        = 0x00b1", // Plus Minus
"pm 0x1b   = 0x00b1", // Plus Minus
"pm!       = 0x00b1", // Plus Minus

"aeq!      = 0x2248", // Aprox Equal
"aeq 0x1b  = 0x2248", // Aprox Equal
"~=        = 0x2248", // Aprox Equal

"!^        = 0x221a", // Radical / Sqare root
"rad!      = 0x221a", // Radical / Sqare root
"rad 0x1b  = 0x221a", // Radical / Sqare root

"++        = 0x2211", // Summation
"sum 0x1b  = 0x2211", // Summation
"sum!      = 0x2211", // Summation

"**        = 0x220f", // Product
"prod 0x1b = 0x220f", // Product
"prod!     = 0x220f", // Product
```

Appendix F: TTS-Unicode Keymap

```
//
// TTS-Unicode input map
//
"unicode+digit1+digit2+digit3+digit4",

"begin unicode",
"u = 0x0",
"U = 0x0",
"0x30 x = 0x0",
"0x30 X = 0x0",
"end unicode",

"begin digit1",
"0x30=0x0000",
"0x31=0x1000",
"0x32=0x2000",
"0x33=0x3000",
"0x34=0x4000",
"0x35=0x5000",
"0x36=0x6000",
"0x37=0x7000",
"0x38=0x8000",
"0x39=0x9000",
"a=0xA000",
"b=0xB000",
"c=0xC000",
"d=0xD000",
"e=0xE000",
"f=0xF000",
"A=0xA000",
"B=0xB000",
"C=0xC000",
"D=0xD000",
"E=0xE000",
"F=0xF000",
"end digit1",

"begin digit2",
"0x30=0x0000",
"0x31=0x0100",
"0x32=0x0200",
"0x33=0x0300",
"0x34=0x0400",
"0x35=0x0500",
"0x36=0x0600",
"0x37=0x0700",
"0x38=0x0800",
"0x39=0x0900",
"a=0x0A00",
"b=0x0B00",
"c=0x0C00",
"d=0x0D00",
"e=0x0E00",
"f=0x0F00",
"A=0x0A00",
```

```
"B=0x0B00",
"C=0x0C00",
"D=0x0D00",
"E=0x0E00",
"F=0x0F00",
"end digit2",

"begin digit3",
"0x30=0x0000",
"0x31=0x0010",
"0x32=0x0020",
"0x33=0x0030",
"0x34=0x0040",
"0x35=0x0050",
"0x36=0x0060",
"0x37=0x0070",
"0x38=0x0080",
"0x39=0x0090",
"a=0x00A0",
"b=0x00B0",
"c=0x00C0",
"d=0x00D0",
"e=0x00E0",
"f=0x00F0",
"A=0x00A0",
"B=0x00B0",
"C=0x00C0",
"D=0x00D0",
"E=0x00E0",
"F=0x00F0",
"end digit3",

"begin digit4",
"0x30=0x0000",
"0x31=0x0001",
"0x32=0x0002",
"0x33=0x0003",
"0x34=0x0004",
"0x35=0x0005",
"0x36=0x0006",
"0x37=0x0007",
"0x38=0x0008",
"0x39=0x0009",
"a=0x000A",
"b=0x000B",
"c=0x000C",
"d=0x000D",
"e=0x000E",
"f=0x000F",
"A=0x000A",
"B=0x000B",
"C=0x000C",
"D=0x000D",
"E=0x000E",
"F=0x000F",
"end digit4"
```

Appendix G: Useful Unicode Symbols

A list of useful Unicode symbols that can be mapped to either an Latin-1 (Set 1) or a Symbol (Set 2) character.

Unicode Char	Symbol	Description	Motif		Notes
			Set	Char	
Greek Symbols					
0x0391	Α	Capital Alpha	2	0x41	*
0x0392	Β	Capital Beta	2	0x42	*
0x0393	Γ	Capital Gamma	2	0x47	
0x0394	Δ	Capital Delta	2	0x44	
0x0395	Ε	Capital Epsilon	2	0x45	*
0x0396	Ζ	Capital Zeta	2	0x5a	*
0x0397	Η	Capital Eta	2	0x48	*
0x0398	Θ	Capital Theta	2	0x51	
0x0399	Ι	Capital Iota	2	0x49	*
0x039a	Κ	Capital Kappa	2	0x4b	*
0x039b	Λ	Capital Lambda	2	0x4c	
0x039c	Μ	Capital Mu	2	0x4d	*
0x039d	Ν	Capital Nu	2	0x4e	*
0x039e	Ξ	Capital Xi	2	0x58	
0x039f	Ο	Capital Omicron	2	0x4f	*
0x03a0	Π	Capital Pi	2	0x50	
0x03a1	Ρ	Capital Rho	2	0x42	*
0x03a3	Σ	Capital Sigma	2	0x53	
0x03a4	Τ	Capital Tau	2	0x54	*
0x03a5	Υ	Capital Upsilon	2	0x55	
0x03a6	Φ	Capital Phi	2	0x46	

Unicode Char	Symbol	Description	Motif		Notes
			Set	Char	
0x03a7	Χ	Capital Chi	2	0x43	*
0x03a8	Ψ	Capital Psi	2	0x59	
0x03a9	Ω	Capital Omega	2	0x57	
0x03b1	α	Small Alpha	2	0x61	
0x03b2	β	Small Beta	2	0x62	
0x03b3	γ	Small Gamma	2	0x67	
0x03b4	δ	Small Delta	2	0x64	
0x03b5	ε	Small Epsilon	2	0x65	
0x03b6	ζ	Small Zeta	2	0x7a	
0x03b7	η	Small Eta	2	0x68	
0x03b8	θ	Small Theta	2	0x71	
0x03b9	ι	Small Iota	2	0x69	
0x03ba	κ	Small Kappa	2	0x6b	
0x03bb	λ	Small Lambda	2	0x6c	
0x03bc	μ	Small Mu	2	0x6d	
0x03bd	ν	Small Nu	2	0x6e	
0x03be	ξ	Small Xi	2	0x78	
0x03bf	ο	Small Omicron	2	0x6f	
0x03c0	π	Small Pi	2	0x70	
0x03c1	ρ	Small Rho	2	0x72	
0x03c2	ς	Small Final Sigma	2	0x56	
0x03c3	σ	Small Sigma	2	0x73	
0x03c4	τ	Small Tau	2	0x74	
0x03c5	υ	Small Upsilon	2	0x75	
0x03c6	φ	Small Phi	2	0x66	
0x03c7	χ	Small Chi	2	0x63	
0x03c8	ψ	Small Psi	2	0x79	
0x03c9	ω	Small Omega	2	0x77	
0x03d2	ϒ	Upsilon with Hook	2	0xa1	
0x03d6	ϖ	Small Omega Pi	2	0x76	
0x03d1	ϑ	Small Script Theta	2	0x4a	
0x03d5	φ	Small Script Phi	2	0x6a	

Unicode Char	Symbol	Description	Motif		Notes
			Set	Char	
Letter-like Symbols					
0x2107	ℇ	Euler constant	2	0x65	!
0x2111	ℑ	Black-letter capital I	2	0xc1	
0x2118	℘	Script capital P	2	0xc3	
0x211c	ℜ	Black-letter capital R	2	0xc2	
0x2122	™	Trade mark sign	2	0xe4	
0x212b	Å	Angstrom sign	1	0xc5	!
0x2135	ℵ	Alef symbol	2	0xc0	
Arrow Symbols					
0x2190	←	Leftwards arrow	2	0xac	
0x2191	→	Rightwards arrow	2	0xad	
0x2192	↓	Downwards arrow	2	0xae	
0x2193	↑	Upwards arrow	2	0xaf	
0x2194	↔	Left-right arrow	2	0xab	
0x21b5	↕	Up-down arrow	1	0xbf	!
0x21d0	⇐	Leftwards double arrow	2	0xdc	
0x21d1	⇑	Upwards double arrow	2	0xdd	
0x21d2	⇒	Rightwards double arrow	2	0xde	
0x21d3	⇓	Downwards double arrow	2	0xdf	
0x21d4	⇔	Left-right double arrow	2	0xdb	
Math Symbols					
0x2200	∀	For All	2	0x22	
0x2202	∂	Partial Differential	2	0xb6	
0x2203	∃	There Exists	2	0x24	
0x2205	∅	Empty Set	2	0xc6	*
0x2206	Δ	Increment / Delta	2	0x44	!
0x2207	∇	Nabla / Del	2	0xd1	
0x2208	∈	Element Of	2	0xce	
0x2209	∉	Not Element Of	2	0xcf	
0x220b	∋	Contains As Element	2	0x27	
0x220f	∏	N-ary Product	2	0xd5	
0x2211	∑	N-ary Summation	2	0xe5	

Unicode Char	Symbol	Description	Motif		Notes
			Set	Char	
0x2213	±	Minus or Plus sign	2	0xb1	*
0x2215	⁄	Division Slash	2	0xa4	*
0x2216	\	Set Mius / Backslash	1	0x5c	!
0x2217	*	Astersk Operator	2	0x2a	*
0x2218	°	Ring operator (Degree sign)	2	0xb0	*
0x2219	•	Bullet operator	2	0xb7	
0x221a	√	Square Root	2	0xd6	
0x221d	∝	Proportional To	2	0xb5	
0x221e	∞	Infinity	2	0xa5	
0x2220	∠	Angle	2	0xd0	
0x2223	|	Divides / Such That	2	0x7c	*
0x2227	∧	Logical And	2	0xd9	
0x2228	∨	Logical Or	2	0xda	
0x2229	∩	Intersection	2	0xc7	
0x222a	∪	Union	2	0xc8	
0x222b	∫	Integral	2	0xf2	
0x2234	∴	Therefore	2	0x5c	
0x2236	:	Ratio (colon)	2	0x3a	*
0x223c	~	Tilde Operator	2	0x7e	*
0x2245	≅	Approximately Equal To	2	0x40	
0x2248	≈	Almost Equal To	2	0xbb	
0x2260	≠	Not Equal To	2	0xb9	
0x2261	≡	Identical To	2	0xba	
0x2264	≤	Less-than Or Equal To	2	0xa3	
0x2265	≥	Greater-than Or Equal To	2	0xb3	
0x226a	«	Much Less-than	1	0xab	!
0x226b	»	Much Greater-than	1	0xbb	!
0x227a	⟨	Precedes	2	0xe1	
0x227b	⟩	Succeeds	2	0xf1	
0x2282	⊂	Subset Of	2	0xcc	
0x2283	⊃	Superset Of	2	0xc9	
0x2284	⊄	Not a Subset Of	2	0xcb	

Unicode Char	Symbol	Description	Motif		Notes
			Set	Char	
0x2286	⊆	Subset Of Or Equal To	2	0xcd	
0x2287	⊇	Superset Of Or Equal To	2	0xca	
0x2295	⊕	Circled Plus	2	0xc5	
0x2297	⊗	Circled Times	2	0xc4	
0x22a5	⊥	Up Tack	2	0x5e	
0x22c4	◊	Diamond Operator	2	0xe0	
0x22c5	·	Dot Operator	2	0xd7	*
0x22d5	#	Equal And Parallel To	2	0x23	*
0x22ef	…	Midline Horizontal Ellipsis	2	0xbc	

* Although a distinctive symbol, it may be confused with another, similar one

! The same mapping is used elsewhere (this is the secondary mapping)

Appendix H: Standard TIM Configuration File

```
##############################################################################
##############################################################################
## This is the Table Tools System (TTS) Table Input Method (TIM) configuration
## file.  Modify at will, but be careful what you're changing.
##
## Notes:
## -----
## 1. Each line in this file can have up to 1023 characters of text
##
## 2. The '\' (backslash) character is the end-of-line escape character -
##    the end-of-line is ignored, and the next line is assumed to be the
##    continuation of the current line - the contents of the two lines are
##    concatenated into a single line. There is no limitation on how many
##    lines you may thus concatenate (this overcomes the limitation of
##    1023 characters per line).
##
## 3. When specifying entries and their values, white space around
##    '=' is removed. As is the white space around separators ',' and ';'
##
## 4. You may get values of  environment variables by de-referencing them
##    with ${VAR} (where VAR is the environment variable name).
##
## 5. You can use '\' as escape char for ASCII characters '\n', '\t', '\b',
##    and to escape space char ' ' and '\' itself.
##############################################################################
##############################################################################

##############################################################################
## Standard configuration of the Yudit component
##############################################################################
[Default]

Path=${HOME}:${TTS_HOME}/yudit
Encoding=UTF8
Input=TTS-input
XInput=None

Font=Unicode
Size=15
Slant=Any
Spacing=Any
Weight=Any

Add.Style=Any
Avg.Width=Any

Language=English
Tab Size=8
Foreground=black
Background=white
Cursor=red

Printer=Default

[Menu Item]

Printer=Default,Preview,Untitled.ps

Encoding=UTF8,UNI
Input=Straight,TTS-input,MY-Unicode,TTS-Unicode
XInput=None
```

```
Font=English,Unicode,Times,Courier

Language=English

Weight=Any,Regular,Medium,Demibold,Bold
Size=13,15,18,20
Slant=Any,Roman,Italic,Oblique
Spacing=Any,Monospace,CharCell,Proportional

#
# Added from version 1.4
#
Add.Style=Any,sans,serif,fantizi,jiantizi,ja
Avg.Width=Any,50,60,70,80,90,100,110,120

Tab Size=2,4,8

Background=black,white,darkGray,gray,lightGray,red,green,blue,cyan,magenta,yello
w,darkRed,darkGreen,darkBlue,darkCyan,darkMagenta,darkYellow

Foreground=black,white,darkGray,gray,lightGray,red,green,blue,cyan,magenta,yello
w,darkRed,darkGreen,darkBlue,darkCyan,darkMagenta,darkYellow

Cursor=black,white,darkGray,gray,lightGray,red,green,blue,cyan,magenta,yellow,da
rkRed,darkGreen,darkBlue,darkCyan,darkMagenta,darkYellow

[Font Map English]

Font Count=1
Font1=8859_1:*-*-iso8859-1

[Font Map Unicode]

Font Count=1
Font1=10646:misc-fixed-iso10646-1

[Font Map Times]

Font Count=2
Font1=8859_1:adobe-times-iso8859-1
Font2=8859_1:adobe-times-iso8859-2

[Font Map Courier]

Font Count=2
Font1=8859_1:adobe-courier-iso8859-1
Font2=8859_1:adobe-courier-iso8859-2

[Printer Default]

Command=/usr/bin/lpr

[Printer Preview]
Command=cat > ${HOME}/.preview.ps ; gv ${HOME}/.preview.ps &

[Printer Untitled.ps]
File=${HOME}/Untitled.ps

[XInput None]
Version=0.0
Input Type=None
Input Style=None
Font Map=None
Encoding=None
Text Type=None

[Language English]
Short Name=en.utf8
Text Font=English
Text Size=16
Text Weight=Medium
Text Slant=Any
Text Spacing=Any
```

```
Button Font=English
Button Size=16
Button Weight=Bold
Button Slant=Any
Button Spacing=Any


##############################################################################
##############################################################################
##
## Declaration of TTS fonts
##
##############################################################################
##############################################################################
[TIM TTS Fonts]

## Default TTS font.
DefaultFont = -*-times-*-r-*--*-120-*

## First, list all the fonts used within the TTS. Each font entry in the list
## is identified as a combination of the font family name (e.g. Times) and
## font face. The values of font family name and face are combined using a
## hyphen to produce a font entry. The allowed values for the font face are:
##
##   0 - Normal
##   1 - Italic
##   2 - Bold
##   3 - Italic-Bold
##
## Note that for each font entry the font family name can contain white space.
##
## For example, suppose TTS uses only the Times font family with all the faces
## of this font, then, we would list the fonts as follows:
##
## Fonts=Times-0,Times-1,Times-2,Times-3
##

Fonts=Times-0,Times-1,Times-2,Times-3,Symbol-0

## Next, for each font entry declared, specify the X font, in the X Logica
## Font Description format, these font entries map to. For example:
##
## Times-0=-*-times-*-r-*--*-120-*
##
## You can use ‘xfontsel’ utility to choose the X font.

Times-0     = -*-times-*-r-*--*-120-*
Times-1     = -*-times-*-i-*--*-120-*
Times-2     = -*-times-bold-r-*--*-120-*
Times-3     = -*-times-bold-i-*--*-120-*
Helvetica-0 = -*-helvetica-*-r-*-*-*-120-*
Helvetica-1 = -*-helvetica-*-o-*-*-*-120-*
Helvetica-2 = -*-helvetica-bold-r-*-*-*-120-*
Helvetica-3 = -*-helvetica-bold-o-*-*-*-120-*
Courier-0   = -*-courier-*-r-*-*-*-120-*
Courier-1   = -*-courier-*-o-*-*-*-120-*
Courier-2   = -*-courier-bold-r-*-*-*-120-*
Courier-3   = -*-courier-bold-o-*-*-*-120-*
Symbol-0    = -*-symbol-*-*-*-*-120-*

######################################################################
######################################################################
##
## Declaration of parameters needed to locate the TIM parsers.
##
######################################################################
######################################################################
[TIM Java Parsers]

##
## Classpath in addition to the default ${TTS_HOME}
##
```

```
CLASSPATH=

Parsers = parser1, parser2
#Parsers=parser2,parser1

parser1 = Infix Grammar, tim.grammars.g1.TTSGrammarParser
parser2 = Prefix Grammar, tim.grammars.g2.PrefixParser

ParserDefaultIndex = 0



################################################################################
################################################################################
##
## Table Templates
##
################################################################################
################################################################################
[TIM Table Templates]

################################################################################
## Table layout specification for tables with at most 2-dimensional grids
################################################################################
##
## For each table template specify number of grids, grid placement, and grid
## dimension ranges.
##
## Girds are placed in a container that organizes them in ordered rows, called
## grid-lines, and columns, called grid-stack. Grids can be indexed by their
## {grid-line, grid-stack} position in the container.
##
################################################################################

TableTemplates = Normal-UpDown, Normal-LeftRight, Inverted, Vector, Decision

################################################################################
## Normal Table - Up-Down case
################################################################################

##
## Table CCG Cases:
## NORMAL   = 0
## INVERTED = 1
## VECTOR   = 2
## DECISION = 3
##
Normal-UpDown.ccgCase = 0

##
## Specify the dimension of the container of grids.
##
Normal-UpDown.container_dimension = 2, 2

##
## Specify where various girds are placed using grid symbol id. Value of E
## indicates that the cell does not contain any grid of the table, and will be
## empty when displayed.
##

Normal-UpDown.grid_placement = \
   E, H2; \
  H1,  G;

## Specify girds dimensions. This is either 0, 1, or 2. Value of 1 means
## single row or column, and 2 - multiple rows and columns. Value of 0
## indicates that the cell does not have a dimension - this corresponds with
## value of E in the grid_plcement.
##
## Ensure that the grid dimensions are consistent with the number of rows and
## columns boundaries set in the grid_lines_min_rows, grid_lines_max_rows,
## grid_stacks_min_columns, and grid_stacks_max_columns variables. For
## example, a grid of dimension 1 should not be allowed to have both row and
```

```
## column max values to be greater than 1 (since then it would be 2
## dimensional).
Normal-UpDown.grid_dimensions = \
   0,  1; \
   1,  2;

##
## Here we specify the permitted ranges of the dimensions of grids. For each
## grid-line (grid-stack) we specify the minimum and maximum number of rows
## (columns) permitted. The user will be limited to indicate the number of
## rows (columns) per grid-line (gird-stack) within the range specified
## here. The user's choice of number of rows (columns) will be enforced across
## all grids within the grid-line (grid-stack). Thus, the grid dimensions will
## remain consistent with each other.
##
## For example, the final table will look something like this (container
## boundaries are delimited by symbols 'I' and '=', grids are delimited by
## symbols '|' and '-'):
##
## For example:
##
##    stack 1        stack 2              stack 3
## ========================================================
## I -------- I -------------- I ---------------------- I
## I |      | I |      |      | I |      |      |      | I
## I -------- I -------------- I ---------------------- I   line 1
## I |      | I |      |      | I |      |      |      | I
## I -------- I -------------- I ---------------------- I
## ========================================================
## I -------- I -------------- I ---------------------- I
## I |      | I |      |      | I |      |      |      | I
## I -------- I -------------- I ---------------------- I
## I |      | I |      |      | I |      |      |      | I   line 2
## I -------- I -------------- I ---------------------- I
## I |      | I |      |      | I |      |      |      | I
## I -------- I -------------- I ---------------------- I
## ========================================================
## I -------- I -------------- I ---------------------- I
## I |      | I |      |      | I |      |      |      | I
## I -------- I -------------- I ---------------------- I
## I |      | I |      |      | I |      |      |      | I
## I -------- I -------------- I ---------------------- I   line 3
## I |      | I |      |      | I |      |      |      | I
## I -------- I -------------- I ---------------------- I
## I |      | I |      |      | I |      |      |      | I
## I -------- I -------------- I ---------------------- I
## ========================================================
##
## For all the grid-lines, specify the allowed range of number of rows
## permitted in the grids. TIM will enforce the actual number of rows used to
## be one within that range (all the grids in the grid-line will have the same
## number of rows).
##
## For the above example these values could be:
##
## Normal-UpDown.grid_lines_min_rows = 2, 1, 1
## Normal-UpDown.grid_lines_max_rows = 2, 3, MAX
##
Normal-UpDown.grid_lines_min_rows = 1, 1
Normal-UpDown.grid_lines_max_rows = 1, MAX

##
## Similarly, for all the grid-stacks, specify the allowed range of number of
## columns permitted in the grids. TIM will enforce the actual number of
## columns used to be one within that range (all the grids in the grid-stack
## will have the same number of columns).
##
## For the above example these values could be:
##
## Normal-UpDown.grid_stacks_min_columns=1, 1, 1
## Normal-UpDown.grid_stacks_max_columns=1, 2, MAX
##
Normal-UpDown.grid_stacks_min_columns = 1, 1
```

```
Normal-UpDown.grid_stacks_max_columns = 1, MAX

## Set the minimum and default sizes of the GUI grid components. The user
## cannot resize the grids beyond the minimum values set using
## grid_lines_min_heights and grid_stacks_min_widths. Initial dimensions are
## set using grid_lines_widths and grid_stacks_heights.

Normal-UpDown.grid_lines_min_heights =   0,  80
Normal-UpDown.grid_lines_heights     =  80, 110

Normal-UpDown.grid_stacks_min_widths =   0, 150
Normal-UpDown.grid_stacks_widths     = 130, 200

################################################################################
## Normal Table - Up-Down case
################################################################################
Normal-LeftRight.ccgCase = 0
Normal-LeftRight.container_dimension = 2, 2
Normal-LeftRight.grid_placement = \
   E, H1; \
  H2,  G;
Normal-LeftRight.grid_dimensions = \
   0,  1; \
   1,  2;

Normal-LeftRight.grid_lines_min_rows =     1, 1
Normal-LeftRight.grid_lines_max_rows =     1, MAX
Normal-LeftRight.grid_stacks_min_columns = 1, 1,
Normal-LeftRight.grid_stacks_max_columns = 1, MAX

Normal-LeftRight.grid_lines_min_heights =  0,  80
Normal-LeftRight.grid_lines_heights     = 80, 110

Normal-LeftRight.grid_stacks_min_widths =   0, 150
Normal-LeftRight.grid_stacks_widths     = 130, 200

################################################################################
## INVERTED TABLE
##
## Need to cover 3 cases:
##
## 1.
##                 --------
##                 |      |
##                 |  H2  |
##                 |      |
##                 --------
##                    ^
##                    |
##                    |
##  --------       --------
##  |      |       |      |
##  |  H1  | ---> |  G   |
##  |      |       |      |
##  --------       --------
##
##
## 2.
##                 --------
##                 |      |
##                 |  H2  |
##                 |      |
##                 --------
##                    |
##                    |
##                    V
##  --------       --------       --------
##  |      |       |      |       |      |
##  |  H1  | <--- |  G   | ---> |  H3  |
##  |      |       |      |       |      |
##  --------       --------       --------
##
## 3.
```

```
##                --------
##                |      |
##                |  H2  |
##                |      |
##                --------
##                   ^
##                   |
##                   |
##  --------      --------
##  |      |      |      |
##  |  H1  | <--- |  G   |
##  |      |      |      |
##  --------      --------
##
##
##  If you ignore the arrows, for the moment, you will notice
##  all of the tables have the placement of grids G, H1 and H2
##  In common. The case (2) has an extra grid.
##
##  Hence, our grid placement layout will be the case (2). Case
##  (1) and (3) will assume that H3's dimension will be 0, and
##  hence it will not be shown.
##
################################################################################

Inverted.ccgCase = 1
Inverted.container_dimension = 2, 3
Inverted.grid_placement = \
   E, H2,  E; \
  H1,  G, H3;
Inverted.grid_dimensions = \
   0,  1,  0; \
   1,  2,  1;
Inverted.grid_lines_min_rows =     1, 1
Inverted.grid_lines_max_rows =     1, MAX
Inverted.grid_stacks_min_columns = 1, 1,   0
Inverted.grid_stacks_max_columns = 1, MAX, 1

Inverted.grid_lines_min_heights =  0,  80
Inverted.grid_lines_heights     = 80, 110

Inverted.grid_stacks_min_widths =   0, 150,  0
Inverted.grid_stacks_widths     = 130, 200, 80

################################################################################
## VECTOR TABLE
################################################################################
Vector.ccgCase = 2
Vector.container_dimension = 2, 2
Vector.grid_placement = \
   E, H2;\
  H1,  G;
Vector.grid_dimensions = \
   0,  1; \
   1,  2;
Vector.grid_lines_min_rows =     1, 1
Vector.grid_lines_max_rows =     1, MAX
Vector.grid_stacks_min_columns = 1, 1
Vector.grid_stacks_max_columns = 1, MAX

Vector.grid_lines_min_heights =  0,  80
Vector.grid_lines_heights     = 80, 110

Vector.grid_stacks_min_widths =   0, 150
Vector.grid_stacks_widths     = 130, 200

################################################################################
## DECISION TABLE
################################################################################
Decision.ccgCase = 3
Decision.container_dimension = 2, 2
Decision.grid_placement = \
   E, H2; \
```

```
  H1,  G;
Decision.grid_dimensions = \
   0,  1; \
   1,  2;
Decision.grid_lines_min_rows =     1, 1
Decision.grid_lines_max_rows =     1, MAX
Decision.grid_stacks_min_columns = 1, 1
Decision.grid_stacks_max_columns = 1, MAX

Decision.grid_lines_min_heights =   0,  80
Decision.grid_lines_heights     =  80, 110

Decision.grid_stacks_min_widths =   0, 150
Decision.grid_stacks_widths     = 130, 200

##############################################################################
##############################################################################
##
## Declaration of Unicode to ASCII/Symbol character mapping
##
##############################################################################
##############################################################################
[TIM Unicode Map]

##
## Note: all use of hexadecimal notation must be in LOWER case.
##

##############################################################################
## Variable DefMap specifies default unicode mapping. If the Unicode character
## is not listed below, it will be mapped to this default value. The
## upside-down '?'  is a good choice (character 0xbf in ASCII).
##############################################################################

DefMap=0x01 0xbf

##############################################################################
## Indicate what Unicode character ranges we are interested in. For now, we
## care only for *RENDERABLE NON-WHITE-SPACE SUBSETS* of Letter-Like Symbols
## (0x2100-0x214f), Arrows (0x2190-0x21ff) and Mathematical Operators
## (0x2200-0x22ff). We care only for the characters we can actually render,
## so the ranges indicated are subsets of actual sets we're interested in.
##
## For detailed description of Unicode characters, see the code charts at the
## Unicode Consortium site at http://www.unicode.org.
##
## Note that you should not specify the *RENDERABLE NON-WHITE-SPACE SUBSETS*
## of Latin-1 set (0x0000-0x00FF) - the valid subsets 0x0021-0x007e and
## 0x00a1-0x00ff are predefined in TIM by default, (see TIMConfig.cpp,
## ranges_[0] and ranges_[1]).
##############################################################################

Ranges=0x2100-0x213a,0x2190-0x21f3,0x2200-0x22f1

##############################################################################
## List all the desirable mappings from a Unicode character to an ASCII or a
## Symbol character.  Each entry is the unicode character to be mapped, and its
## value is composed of the character set id and the character in the set
## separated by empty space.
##
## The unicode characters listed here must fall within one the ranges indicated
## in the 'Ranges' variable.  Not all unicode characters in each range must be
## declared - only those that can be mapped. The unicode characters that
## omitted will mapped to a default value indicated by the 'defMap' variable.
##
## Format:
##
## UnicodeCharInHex = [0x01|0x02] charIdInHex
##
## where:
##
##  0x01 = ASCII
##  0x02 = Symbol
```

```
##
## Example: the FOR_ALL symbol is a unicode character 0x2200. It maps to
## character 0x22 in the Symbol character set. Therefore, the corresponding map
## entry would like as follows:
##
## 0x2200=0x02 0x22
##
## Notes:
##
## 1. All unicode characters must be specified in hexadecimal notation with
## 4 digits following '0x'. All hex characters (0-9, a-f & x) must be in
## *LOWER* case. That is, none of the following unicode values are correct:
##
## 0x322  - only 3 digits!
## 0X2203 - used X instead of x
## 0X22FF - used X and F instead of x and f
## 0x22FF - used F instead of f
## 0x22fF - (***tricky***) used F instead of f (the last one!)
##
## 2. Mapping unicode characters in the ASCII range (0x0000-0x00ff) has
## no effect.  Unicode and ASCII map 1-to-1 up to 0x00ff, and the user cannot
## change that.


## Euler constant, mapped to Greek small letter Epsilon
0x2107=0x02 0x65

## Black-letter capital I
0x2111=0x02 0xc1

## Script capital P
0x2118=0x02 0xc3

## Black-letter capital R
0x211c=0x02 0xc2

## Trade mark sign, similar to another Symbol Trademark 0xd4
0x2122=0x02 0xe4

## Angstrom symbol, mapped to Latin Angstrom letter
0x212b=0x01 0xc5

## Alef symbol
0x2135=0x02 0xc0

## Leftwards arrow
0x2190=0x02 0xac

## Rightwards arrow
0x2191=0x02 0xad

## Downwards arrow
0x2192=0x02 0xae

## Upwards arrow
0x2193=0x02 0xaf

## Left-right arrow
0x2194=0x02 0xab

## Up-down arrow, mapped to Latin Up-Down symbol
0x21b5=0x01 0xbf

## Leftwards double arrow
0x21d0=0x02 0xdc

## Upwards double arrow
0x21d1=0x02 0xdd

## Rightwards double arrow
0x21d2=0x02 0xde

## Downwards double arrow
```

```
0x21d3=0x02 0xdf

## Left-right double arrow
0x21d4=0x02 0xdb

## For All
0x2200=0x02 0x22

## Partial Differential
0x2202=0x02 0xb6

## There Exists
0x2203=0x02 0x24

## Empty Set, similar to Latin Empty Set symbol 0xd8
0x2205=0x02 0xc6

## Increment / Delta, mapped to Greek letter Delta
0x2206=0x02 0x44

## Nabla / Del
0x2207=0x02 0xd1

## Element Of
0x2208=0x02 0xce

## Not Element Of
0x2209=0x02 0xcf

## Contains As Element
0x220b=0x02 0x27

## N-ary Product
0x220f=0x02 0xd5

## N-ary Summation
0x2211=0x02 0xe5

## Minus or Plus sign, mapped to Latin Plus or Minus 0xb1
0x2213=0x02 0xb1

# Division Slash, similar to Latin Slash 0x2f, similar to Symbol Slash 0x2f
0x2215=0x02 0xa4

## Set Mius, mapped as Latin Backslash
0x2216=0x01 0x5c

## Asterisk Operator, similar to Latin Asterisk 0x2a
0x2217=0x02 0x2a

## Ring operator, similar to Latin Degree symbol 0xb0
0x2218=0x02 0xb0

## Bullet operator
0x2219=0x02 0xb7

## Square Root
0x221a=0x02 0xd6

## Proportional To
0x221d=0x02 0xb5

## Infinity
0x221e=0x02 0xa5

## Angle
0x2220=0x02 0xd0

## Divides / Such That, Similar to Latin 0x7c
0x2223=0x02 0x7c

## Logical And
0x2227=0x02 0xd9
```

```
## Logical Or
0x2228=0x02 0xda

## Intersection
0x2229=0x02 0xc7

## Union
0x222a=0x02 0xc8

## Integral
0x222b=0x02 0xf2

## Therefore
0x2234=0x02 0x5c

## Ratio (colon), Similar to Latin 0x3a
0x2236=0x02 0x3a

## Tilde Operator, similar to Latin 0x7e
0x223c=0x02 0x7e

## Approximately Equal To
0x2245=0x02 0x40

## Almost Equal To
0x2248=0x02 0xbb

## Not Equal To
0x2260=0x02 0xb9

## Identical To
0x2261=0x02 0xba

## Less-than Or Equal To
0x2264=0x02 0xa3

## Greater-than Or Equal To
0x2265=0x02 0xb3

## Much Less-than, mapped to Latin Left-Pointing Double Angle Quotation Mark
0x226a=0x01 0xab

## Much Greater-than, mapped to Latin Right-Pointing Double Angle Quotation Mark
0x226b=0x01 0xbb

## Precedes
0x227a=0x02 0xe1

## Succeeds
0x227b=0x02 0xf1

## Subset Of
0x2282=0x02 0xcc

## Superset Of
0x2283=0x02 0xc9

## Not a Subset Of
0x2284=0x02 0xcb

## Subset Of Or Equal To
0x2286=0x02 0xcd

## Superset Of Or Equal To
0x2287=0x02 0xca

## Circled Plus
0x2295=0x02 0xc5

## Circled Times
0x2297=0x02 0xc4
```

```
## Up Tack
0x22a5=0x02 0x5e

## Diamond Operator
0x22c4=0x02 0xe0

## Dot Operator, similar to Latin 0xd7
0x22c5=0x02 0xd7

## Equal And Parallel To, similar to Latin 0xd7 (hash symbol)
0x22d5=0x02 0x23

## Midline Horizontal Ellipsis (mapped to Horizontal Ellipsis)
0x22ef=0x02 0xbc

##############################################################################
##############################################################################
##
## Info Display Format
##
##############################################################################
##############################################################################
[TIM Info Display Format]

InfoClasses = Default, Name, FontFamily, Font-Face, Tag, FontCode, Arity, \
            Rules, CCG, C_DataType, Domain, CType, CForm, IDP_Inst, IDP_Card, \
            IDP_I, IDP_G, IDP_Q, Form, Parent, MapleName, MapleDefn, Unicode

Default.type        = enum
Default.numChar     = 7
Default.enumVal     = No, Yes

Name.type           = string
Name.numChar        = 16

FontFamily.type     = string
FontFamily.numChar  = 10
FontFamily.strList  = Times, Helvetica, Courier, Symbol

Font-Face.type      = enum
Font-Face.numChar   = 11
Font-Face.enumVal   = Normal, Italic, Bold, Italic-Bold

Tag.type            = enum
Tag.numChar         = 9
Tag.enumVal         = Const, PConst, Var, FA, PE, LE, QLE, FTable, PdTable, \
                      Placeholder

FontCode.type       = decimal
FontCode.numChar    = 8

Arity.type          = decimal
Arity.numChar       = 5

Rules.type          = string
Rules.numChar       = 10

CCG.type            = enum
CCG.numChar         = 8
CCG.enumVal         = Normal, Inverted, Vector, Decision

C_DataType.type     = binary
C_DataType.numChar  = 8

Domain.type         = decimal
Domain.numChar      = 8

CType.type          = string
CType.numChar       = 8

CForm.type          = string
CForm.numChar       = 8
```

```
IDP_Inst.type       = string
IDP_Inst.numChar    = 8

IDP_Card.type       = decimal
IDP_Card.numChar    = 8

IDP_I.type          = string
IDP_I.numChar       = 8

IDP_G.type          = decimal
IDP_G.numChar       = 5

IDP_Q.type          = decimal
IDP_Q.numChar       = 5

Form.type           = string
Form.numChar        = 8

Parent.type         = decimal
Parent.numChar      = 6

MapleName.type      = string
MapleName.numChar   = 11

MapleDefn.type      = string
MapleDefn.numChar   = 11

Unicode.type        = hexadecimal
Unicode.numChar     = 7


##############################################################################
##############################################################################
##
## Editor Misc stuff
##
##############################################################################
##############################################################################
[TIM Editor Misc]

## Display string used for items that are not available.
## Note: there is a space char after '\' char.
NotAvailableString = \

## For undefined symbols display, indicate which symbol classes should be
## shown, and in what order (the order is arbitrary, and you may change
## it). You should always show Name, Arity, Tag, Unicode, FontFamily,
## Font-Face.
UndefinedSymbolsShowInfoClasses = Name, Arity, Tag, Unicode, FontFamily, \
                                                        Font-Face

## Values for these classes are required to be entered by the user. You must
## always have Tag, FontFamily and Font-Face since these are required by
## ASTNode.cpp
UndefinedSymbolsRequiredValues = Tag, FontFamily, Font-Face

## For new symbols display, indicate which symbol classes should not be
## allowed to be edited. You must always have Name and Arity since these
## are required by ASTNode.cpp.
UndefinedSymbolsNonEditable = Name, Arity

## For known symbols, indicate which symbol classes should be shown, and
## in what order (the order is arbitrary, and you may change it).
KnownSymbolsInfoClasses = Name, Arity, FontFamily, Font-Face, Tag, CCG, \
                                                Unicode

## For new symbols display, indicate which symbol classes should not be
## allowed to be edited. You must always have Name and Arity since these
## are required by ASTNode.cpp.
KnownSymbolsNonEditable =

## For table symbols display, indicate which symbol classes should be shown,
## and in what order.TableSymbolShowInfoClasses = Name, Tag, CCG
```

```
## For table symbols display, indicate which symbol classes should be shown,
## and in what order.
SyntaxTreeShowInfoClasses = Name, Tag, Arity, Unicode

## Auto parse on at start? (Yes/True, No/False)
AutoParseOn = No
## Auto parse delay values list
AutoParseDelayList = 500, 700, 1000, 2000, 4000
## Auto parse default value list index, 0 is the first value
AutoParseDelayListDefaultIndex = 1
## Maximum auto corrections
MaxAutoCorrections = 8

## Show tooltips? (Yes/True, No/False)
HelpTipsOn = Yes

## Font used in the titles of dialogs and windows
TitleFont = -*-helvetica-bold-r-normal-*-14-140-75-75-*-*-*-*

## Current cells get expanded to show maximal width in the column and maximal
## width in the row. If this value is True/Yes then all other cells will be
## resized to the values indicated by NonCurrentCellsPixelWidth and
## NonCurrentCellsPixelHeight.
ShrinkNonCurrentCells = No
NonCurrentCellPixelWidth = 23
NonCurrentCellPixelHeight = 23

## Grid colours
MainGridBackgroundColor = PowderBlue
MainGridForegroundColor = Black
HeaderGridBackgroundColor = LemonChiffon
HeaderGridForegroundColor = Black

GridNameFont = -*-times-bold-r-*--*-120-*

ContainerShadowThickness = 5

SpecialCellFormatString = <%s>

UndefinedExpnCellString = undef
UndefinedExpnCellEditorMessage = \
This cell is undefined, double-click\n\
on the cell to set the expression type.

TabularExpnCellEditorMessage = \
This cell contains a table, double-click\n\
on the cell to launch the table editor.

NonEditableBackgroundColor = Grey
NonEditableForegroundColor = Black

RowColumnLabelFormatString = %d/%d
RowColumnLabelBackgroundColor = Bisque
RowColumnLabelForegroundColor = Blue

# End of config file
```